The Dissertation Committee for Tyler Scott Hunt
certifies that this is the approved version of the following dissertation:

Private Computation on Public Clouds

Committee:

Emmett Witchel, Supervisor

David Nellans

Christopher J. Rossbach

Hovav Shacham

Private Computation on Public Clouds

by

Tyler Scott Hunt

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August 2020

To Victoria: doing the work requires stealing the time to do it, and I stole most of it from you.

Acknowledgments

Without my collaborators this document would be worse. Without my support system this document would end abruptly, unfinished. Without my advisor this document would be empty.

I've had the great pleasure of working with a bunch of very smart people. I'd first like to acknowledge Yuanzhong Xu, Youngin Kwon, and Sangman Kim for their invaluable early guidance. Talking to someone who knows what they're doing is always valuable, especially when you don't. I'd also like to thank Zhiting Zhu, Zhipeng Jia, Yige Hu, Vance Miller, Simon Peter, and Ariel Szekely all of whom contributed to the work presented here. Finally, a big thank you to my committee members: Christopher J. Rossbach, Hovav Shacham, and David Nellans for their insightful questions and guidance which helped to hone the work to produce this document. Christopher J. Rossbach deserves another mention here since he also got me a job.

I do not believe I would have been able to sustain this seven year effort without a little help from my friends. It's not always fun, and having a solid group of people to look forward to seeing in the margins often made all the difference. A special thank you from the bottom of my crusty, musty heart to the denizens of the Dead Music Capital Band for being a big part of that difference. In that vein, I would like to acknowledge my family and especially

my wife, Victoria, who provide a rock-solid foundation for everything that I do, and this is no exception.

Finally, its impossible to overstate my gratitude to Emmett Witchel, my advisor. Suffice to say he taught me everything I know, so all complaints about the work should be addressed to him.

I am forever humbled to have had the pleasure of walking along this stretch of the path with all of you.

Tyler Hunt

The University of Texas at Austin
August, 2020

Private Computation on Public Clouds

Tyler Scott Hunt, Ph.D.
The University of Texas at Austin, 2020

Supervisor: Emmett Witchel

Public clouds offer valuable services at the expense of privacy. Since the cloud provider controls the privileged software on their machines (the operating system and the hypervisor), they enjoy access to the secrets processed by the applications they host. As a result, users must either trust public clouds or avoid them. Recently, hardware manufacturers have extended CPU designs to provide trusted execution environments (TEEs). Hardware ensures the data inside a TEE can only be accessed by the code inside that TEE, protecting secrets from all software that the provider controls.

However, TEEs do not provide meaningful security for many applications on their own. In practice, many applications are proprietary or make use of accelerators like GPUs. Code inside the TEE has access to user secrets and the freedom to communicate them to the outside world; users cannot vet proprietary code to ensure it does not exercise that freedom (accidentally or intentionally). GPUs are not controlled by the CPU directly but instead by drivers under the cloud provider's control, making it trivial for the cloud

provider to extract secrets that the user offloads to a GPU for processing. GPU TEEs can prevent unauthorized access to GPU memory, but communication with the GPU can still leak information.

We demonstrate system designs that leverage existing (CPU) and proposed (GPU) TEEs that protect users‘ data even when the application code is colluding with the cloud provider to steal it, or when the user offloads parts of the application to GPUs.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Public clouds are collections of computing services offered over the Internet to those willing to pay for their use. Some notable examples of public cloud providers are Amazon Web Services [Amab], Microsoft Azure [azu14], and Google Cloud [Goo]. Public clouds fill two roles: first, they host public-facing services like image editing (Pixlr [Pix]), tax preparation (TurboTax [Int]), or even personal health analyses (23andMe [23ab]). Second, public clouds provide large pools of resources. Users with computationally intensive workloads—e.g., training deep neural networks (DNNs)—can rapidly gain access to a large pool of machines with large amounts of memory, many CPU cores, and accelerators like GPUs. Public clouds have asserted themselves as platforms of consequence; Amazon Web Services alone reported 35 billion dollars in revenue for 2019 (about a third of the public cloud market) [Sta].

The *de facto* success of public clouds is obvious from their market size, but public clouds remain a non-option for users with sensitive data. Cloud providers necessarily control the privileged software (i.e., the hypervisor and the operating system) on their machines for legitimate reasons: controlling privileged software allows providers to multiplex many users across physical machines, ultimately reducing costs and improving user choice. However, a

side effect of that control is that the cloud provider has full access to the state of applications. An application can do very little to hide secret data from the cloud provider. Privileged software controls scheduling, so applications can be interrupted arbitrarily, giving the cloud provider unfettered access to any secrets entrusted to them, no matter how briefly the secrets are in memory.

Cloud users must trust providers to provide integrity and privacy or they must avoid using public clouds altogether. Security-conscious users must reason about the implications of sharing secret data with public cloud providers in addition to the operators of the services themselves. For instance, a user of 23andMe might hesitate to disclose data about their health to a company that might use it for targeted advertisements like Google; a user like Netflix may want to protect the secrets behind their movie recommendation system from Amazon, who runs a competing video service.

This dissertation demonstrates systems that provide secrecy guarantees to public cloud users *without* trusting the platform provider. These systems leverage a hardware isolation mechanism that stops the cloud provider from accessing secrets while preserving much of the application's performance: Trusted Execution Environments. (TEEs). Hardware (i.e., the processor) ensures that only TEE code can access TEE data; hardware protects secrets from all software that the provider controls, including privileged software.

TEEs have found their way into commodity CPUs. Examples of TEEs include Intel SGX [Int14], RISC-V Keystone [LKC+18, LKS+20], and the secure world of ARM TrustZone [Lim]. TEEs have also been proposed for

GPUs [VVB18, JTK+19], though none are available on the market.

While essential to the work described here, TEEs alone are not sufficient to provide meaningful security. One glaring drawback is that prevalent side channels undermine TEE isolation. Intel SGX has accumulated a robust catalog of exploits [XCP15, VBMW+18, GESM17, BCD+18]. While these vulnerabilities certainly call into question the security of TEEs as they exist today, these exploits do not point to fundamental problems with TEEs; in fact, Keystone [LKS+20] designs have already removed many side-channel exploits, e.g., by way-partitioning the last-level cache. Rather than address bugs and oversights that hardware can and should be solve, the work described here focuses on core shortcomings of TEE designs and shows how system software can mitigate them.

We have identified two significant areas where TEEs do not provide meaningful security as designed. First, data-processing services are often closed source. Service providers have incentives to keep their code secret, forcing the user to trust them with secrets to use the service. Second, offloading computation to GPUs leaves applications open to easily exploitable timing attacks.

1.1 Protecting secrets from the services that process them

Operators of user-facing services often keep their code proprietary to protect their competitive advantage: obscuring their service's details creates a

barrier protecting ideas they spent time and money developing. An unfortunate side effect is that users of the service have little visibility into what the code is doing. Any code inside a TEE can access all TEE data— including user secrets—and has the freedom to communicate anything it can access to the outside world. Proprietary code, by definition, cannot be vetted to ensure it does not exercise that freedom (accidentally or intentionally). Furthermore, sometimes the cloud provider *is* the service provider, so collusion between application code in the TEE and the platform is also a concern.

Ryoan [HZX+16, HZX+18] is a system designed to address this problem. Ryoan is a distributed sandbox that allows users to keep their data secret without trusting the software stack, developers, or administrators of these services. The core idea is to confine the code that users cannot vet (i.e., untrusted code). Service code can remain proprietary, but the Ryoan sandbox carefully controls its communication with the outside world, confining the service. Confining untrusted code is a longstanding problem that remains technically challenging [Lam73]. Ryoan meets the challenges of confinement by taking advantage of TEEs and by assuming a request-oriented data model. Confined services only process input once and cannot read or write persistent state (storage) after receiving the input. This model limits Ryoan's applicability to request-oriented server applications—but such servers are the most common way to bring scalable services to large numbers of users.

A naïve (but secure) approach to a confinement system like Ryoan restricts services to a single TEE. This naïve design has performance limitations

because TEEs cannot span multiple processors, and it raises privilege separation concerns since some services involve code and interests from mutually distrustful parties. Rather than resigning applications to those restrictions, Ryoan allows providers to distribute their services across many processors. Ryoan also provides a coarse-grained information flow control mechanism to protect service provider secrets and user secrets when they flow through TEEs containing code that the service providers themselves do not trust. With these mechanisms, Ryoan achieves reasonable flexibility for service providers without sacrificing security.

1.2 Securely offloading computation to cloud GPUs

Performance improvements enabled by GPUs have driven the success of machine learning and computer vision in application domains such as medicine [Hem17, SFB+15], finance [GGKSC13], insurance [NVI16], and communication [NVI17a]. Cloud providers have taken notice; today, GPUs are available on every major public cloud. However, GPU computation is still insecure in public clouds: without hardware intervention, the provider is free to examine GPU state at will and extract secret data.

GPU TEEs (when realized) will protect secrets from cloud providers while they are on GPUs, but offloading computation to a GPU involves communication over the PCI bus[1]. While GPU TEEs protect the *content* of com-

[1]We focus on discrete, PCI-attached GPUs because that is the form factor of the best performing GPUs at time of writing.

munication (e.g., by encryption for Graviton TEEs [VVB18]), they do not protect the *pattern* of communication. Privileged software allows platform providers to observe the pattern of PCI communication. To show the information leaked in GPU communication patterns, we demonstrate an attack on deep neural network image recognition. We trained a classifier on the GPU communication patterns of an image recognition application for two image classes from ImageNet [DDS+09]. Our model was able to distinguish the images of the two classes with 78% accuracy even though it only had access to timing information (never the images themselves).

We designed **Telekine** so that users can securely offload computation to cloud GPUs. Telekine ensures that communication between the user's CPU code and the offloaded GPU computation does not leak by transforming GPU API calls into *data oblivious streams*. Telekine constructs data-oblivious streams by reducing all API calls to a sequence of code execution (`launchKernel`) and data movement (`memcpy`) commands. It then schedules these operations at a fixed rate, possibly creating new operations, or splitting `memcpy` operations into fixed-size pieces. Fixed-sized, fixed-rate communication ensures that any observable patterns are independent of the input data and, therefore, devoid of side-channel information. Fixed-rate communication is not a novel way to eliminate side channels, but Telekine's design shows how to apply it efficiently to modern GPU-based computing.

1.3 Writing conventions and organization

Research contributions are almost always the result of the collaborative effort of many minds. In the spirit of that, I will maintain the convention of using second-person pronouns (our/we) throughout the document. I summarize my specific contributions in footnotes at the beginning of the relevant chapters.

The rest of this document follows this organization. First, we provide a detailed description of the public cloud threat model in Chapter 2, followed by background material on TEEs and GPUs in Chapter 3. Then we describe the designs of Ryoan and Telekine in Chapter 4 and Chapter 5, respectively. Finally, we put the work in context by discussing related work in Chapter 6 and our conclusions in Chapter 7

Chapter 2

The Malicious Public Cloud Threat Model

To completely remove providers of public clouds (platform providers) from all levels of trust, we make the conservative assumption that the platform provider is malicious. The platform provider is a powerful adversary who controls all software executing on the machines that they control. This control extends into privileged software, i.e., the operating system and the hypervisor. By construction, systems that protect data in this model will also protect against weaker adversaries. For instance, the power of attackers who have compromised some part of the platform must be a subset of the cloud provider's power, and it is in the platform provider's power to create fake tenants and mount side-channel attacks.

On the other hand, users must trust hardware and the software of the systems described in this document. These are reasonable requirements since the cloud provider has no control over either of these things. CPUs and GPUs are difficult to modify because their physical packaging is resistant to tampering. Physical modification of these computing devices, especially at scale, would be expensive for a platform provider so it is reasonable to expect them to be deployed without modification. Users must assume that hardware manufacturers are not colluding with platform providers, unless the hardware can

be verified or made open-source for vetting, which is not true of commodity processors. However, hardware providers have little incentive to design insecure hardware: their business model relies on selling processors rather than data. A reputation for an insecure product could hinder processor sales in an increasingly crowded market. The systems described in this dissertation are open-source, allowing users (or communities of users) to vet and validate their implementation, making the assumption of trust weaker.

A caveat to trusting hardware is that there are many examples of failed isolation [HJM+19, HJM+20] (often due to side channels), making much of the current hardware space unfit for purpose. We take the emergence of commercial TEEs as a sign that manufacturers are willing to take security seriously and fix isolation. In the limit, software can be augmented with orthogonal techniques to achieve isolation. There is additional detail about current hardware limitations and known side channels in section 3.2.2.

Denial of service attacks. Denial of service is outside of the scope of our threat model. The platform provider can always leverage their control to refuse to scheduler our system or arbitrarily block communication between the client and the different pieces of our systems.

Chapter 3

Background

Providing security guarantees for computation on an untrusted platform is an active area of research. In almost all cases, users must trust hardware. The exceptions are cryptographic techniques which do not require trust in hardware but greatly magnify the computational and storage costs of the computation [Gen09]. The choice to use a public cloud is an economic one; users can always purchase hardware that they control to run their software securely. Our position is that trusting only hardware is a reasonable compromise, preserving the cloud's economic advantages while providing the building blocks for meaningful security.

The work described here uses trusted CPUs and GPUs, both of which we expect to provide trusted execution environments. Commodity CPUs provide trusted execution environments (TEEs) today. Robust research proposals for GPU TEEs exist, but none have yet made their way into the market.

3.1 Attesting hardware authenticity to a remote user

Hardware hosting computation in a malicious public cloud must be able to prove its authenticity to users. Without proof, the cloud provider can lie

about the presence of secure hardware and steal a user's secret data.

The process of proving that a piece of hardware is genuine is called *attestation*. Specific attestation mechanisms vary, but they all adopt the same general form:

1. The manufacturer embeds a secret in the hardware. The embedding process comes with some reasonable guarantee that malicious actors cannot recover the secret by inspecting the hardware. For instance, a CPU secret might be constructed with physically unclonable properties of the manufactured hardware and never directly exposed to software.

2. Upon request, hardware uses that secret to sign a message cryptographically. This message usually contains some form of nonce from the user so that an attacker cannot reuse messages, and possibly some additional information about hardware state, e.g., SGX signs a description of the program loaded into the TEE.

3. The remote user receives the message and signature, then goes through a validation process. Validation depends on the type of signature; it could be validated against a known public key or forwarded to a trusted service for validation.

3.2 Trusted Execution Environments

Historically, software system designers have leveraged hardware isolation mechanisms as the basis for secure systems. The user/kernel processor

mode bit and page tables on CPUs and GPUs are examples of hardware-enforced isolation mechanisms. These primitives have found favor for decades because they are efficiently implemented in hardware, and have clear semantics that system software can use as the basis for security.

Hardware-supported TEEs are yet another isolation mechanism that is useful for protecting secrets in the malicious public cloud threat model. TEEs are uniquely useful because they separate isolation from resource management (e.g., scheduling on processor cores, and memory management). Other isolation mechanisms allow code to keep secrets; e.g., the user/kernel processor mode bit prevents some code (user-level code) from reading the secrets of others (kernel-level code). However, user-level code cannot keep secrets from kernel-level code. Without an additional mechanism, users cannot keep secrets from the platform provider since they control kernel-level code. TEEs allow the platform provider to remain in control of resources while preventing them from accessing user secrets.

TEE designs have been realized by different hardware vendors for CPUs. Intel has shipped Software Guard eXtensions (SGX) [Int14], which is described in more detail below. ARM's offering is called TrustZone [Lim]. TrustZone provides a "secure world" with provisions for direct control over hardware, but different chip designs vary on how much hardware is under secure world control. Direct control over hardware (e.g., control over the PCI bus) makes secure communication easier because it removes other software's power to observe the communication. However, x86 CPUs are still the predominant plat-

form in public clouds, and it is unclear how compatible secure world control of hardware would be with the other interests of cloud providers. Finally Keystone [LKC+18,LKS+20] is a TEE design for the open-source RISC-V architecture. Keystone has a modular design well suited to selecting the appropriate level of security for a given threat model.

TEE designs have been proposed by researchers for GPUs. Graviton [VVB18] proposes changes to GPU firmware allowing a remote user or a CPU enclave to establish a secure channel for control and data with the GPU. HIX [JTK+19] extends CPU enclaves with trusted MMIO protections to protect communication with the GPU rather than relying on cryptography.

The work in this dissertation builds specifically on Intel Software Guard Extensions [Int14] on CPUs and the proposed Graviton [VVB18] TEEs for GPUs. The remainder of this chapter provides background on those specific TEEs.

3.2.1 Intel Software Guard Extensions

Software Guard Extensions (SGX) [Int14] provide TEEs on recent Intel processors. SGX calls each TEE an *enclave*. An enclave is a region in virtual memory that is protected by hardware. The contents of an enclave are only visible to code that is mapped into the enclave's virtual memory region; this code is said to be *enclave code*. Enclave code can read its enclave and all non-enclave (mapped) memory. Multiple enclaves may exist in a single virtual address space, but they may not read memory from each other's regions. En-

clave code still runs in unprivileged mode (ring 3), and the CPU faults on any instruction that would raise its privilege level.

Memory protection. Processors that support SGX maintain a special physical memory pool called the Enclave Page Cache (EPC). Physical memory can only be mapped into an enclave if it comes from the EPC; complementarily, hardware only allows one enclave to use an EPC page at a time. Data and code can be paged in and out of the EPC by the operating system through purpose-built SGX instructions, but pages are encrypted and MACed by the processor on page-out and then verified and decrypted on page-in.

Memory in the EPC is encrypted by the processor, preventing attackers from stealing plaintext data using bus sniffing attacks. SGX leverages the processor's cache structure to ameliorate the performance overheads involved in encryption: data is is always unencrypted when it is on the chip and is only encrypted and decrypted when it moves to and from main memory.

SGX remote attestation. Attesting an SGX processor to a remote user follows the procedure outlined in Section 3.1 closely. The CPU signs a statement about a particular enclave that the platform provider must forward to the user. The user can validate the statement by querying a trusted attestation service. If validation succeeds, the user can be sure they are communicating with a valid SGX processor.

In addition to proving that the CPU is a real SGX CPU, attestations

serve to bind *identity* to an enclave. For our purposes, it is enough to think of an enclave identity as a hash of the enclave's initial state, i.e., valid memory contents, permissions, and relative position in the enclave. Our trust of the hardware extends to these identities; particularly, we assume that the cloud provider cannot misrepresent the initial state of an enclave under standard cryptographic assumptions. SGX also supports binding enclaves to a specific public.

Knowing the initial state of an enclave allows users to reason about security. An enclave's state can only be mutated by enclave code after initialization. Code that can mutate enclave state must itself be part of the attested initial state.

Hardware threads. CPUs typically have several *hardware threads*. Hardware threads are execution contexts that may execute code concurrently. With respect to SGX, each hardware thread is either in enclave mode or it is not. A thread issues an enclave entry instruction to enter enclave mode. In enclave mode, every instruction must reside on an EPC page that belongs to the enclave that the thread entered. This restriction keeps the operating system from mapping malicious code pages, which would cause the thread to misbehave.

Threads are only allowed to enter enclaves at specifically defined entry points. These points are written into enclave memory (and so are part of the state verified during remote attestation).

If any hardware thread receives an interrupt while in enclave mode, the

processor stores its registers in enclave memory and clears them before the thread is taken out of enclave mode to handle the interrupt. The context can be restored later by another SGX instruction.

3.2.2 Hardware limitations

There are some known security limitations in commodity hardware. We believe the hardware manufacturers should address these limitations (and any additional limitations discovered in the future). Each of these limitations erodes the security afforded by the systems we design to use them. Part of the purpose of building prototype systems is to determine how its security guarantees depend on the security guarantees of the hardware they rely on, thereby motivating fixes for hardware-based limitations.

Transient instruction-based attacks. Transient instructions are processor instructions that are speculatively executed and update the processor's micro-architectural state but are aborted for some reason and do not update the architectural state. Meltdown [LSG+18] and Spectre [KGG+18] are attacks that exploit out of order execution and speculative execution, respectively, to execute transient instructions. These transient instructions influence the processor's micro-architectural state, creating covert channels; both attacks use the processor cache as a proof of concept. Meltdown is specific to Intel processors and allows user-level programs to read arbitrary kernel memory. Spectre applies to a broader range of processors (including Intel, AMD, and ARM), but

is more difficult to exploit. It allows the attacker to read memory belonging to a victim running in a separate address space.

There have been successful Spectre attacks that violate SGX isolation, allowing non-enclave code to read enclave memory [CCX+18, OMA+18].

SGX page faults. On current Intel processors, privileged software can manipulate the page tables of an enclave to observe a page-granularity trace of its code and data. Xu et al. demonstrated attacks that use application-level information to recreate fine-grained secrets from these coarse addresses, e.g., words in a document and object outlines in an image [XCP15].

There is active research on detecting or preventing these attacks using other processor features, e.g., transactional memory [SLKP17, CZRZ17], or monitoring SGX data structures [OTK+18]. If SGX enclaves serviced their own page faults, this leakage channel would disappear.

Address bus monitoring. Although SGX encrypts data in RAM, if an attacker monitors the address bus via a sniffer or a modified RAM chip, it forms a cache line-granularity side or covert channel. No software system can prevent such attacks without new architectural changes.

Processor monitoring. Processor monitoring units (PMUs) provide extensive performance counter information for on-chip events. If the processor updates the PMU about events that occur in enclave-protected execution, the

operating system could use the information as a covert channel to learn secrets via untrusted code, which could modulate its behavior, e.g., to inflate certain event counts.

According to measurements on Skylake processors, the processor disables certain monitoring facilities during enclave execution (e.g., Precise Event-Based Sampling (PEBS)), however the uncore counters (e.g., last level cache misses) are enabled [CD16]. Effective attacks based on branch history have been demonstrated [LSG+17]. It is unknown at this time how effective other attacks based on processor monitoring will be.

Cache timing. Two processes resident on the same core can use cache timing to obtain fine-grained information about each other. For instance, Zhang et al. (on an Amazon-EC2-like platform) extracted ElGamal keys from a non-colluding VM [ZJRR12]. The problem is worse when processes can collude; others have demonstrated high-bandwidth covert channels using cache behavior [XBJ+11, WX15]. There are hardware proposals to address cache timing attacks [LWL15].

3.3 GPUs

Current GPU software stacks prioritize high performance and programmer convenience over security. While a given instance might, at times, have exclusive access to a physical GPU, the provider can migrate instances, exposing GPU state. Modern conveniences like elastic GPUs [Amaa, aws] make

GPUs available over a network connection.

3.3.1 PCIe and device communication

At the hardware level, GPUs can be connected to the system directly on the CPU memory interconnect (integrated) or by the PCIe bus (discrete). PCIe-attached GPUs are overwhelmingly preferred in performance-focused settings because this organization enables the GPU to implement a separate memory subsystem using techniques that enable dramatically higher memory bandwidth. Memory bandwidth is the first-order determinant for performance for most GPU workloads. For example, current NVIDIA cards tout a peak memory bandwidth over 600 GB/s [rtx18], several times higher than the best current peak CPU memory bandwidth. Therefore, we focus our attention on PCIe-attached GPUs, as this is the dominant platform for performance.

The PCIe interfaces provide two forms of communication: memory-mapped I/O (MMIO) and direct memory access (DMA). MMIO is used to re-purpose regions of the physical memory address space for device communication. Contiguous physical ranges, or *BARs* (base-address-regions), are reserved by the hardware, and hardware transparently redirects loads and stores to those regions to the device. MMIO is implemented by configuring CPU registers with metadata describing a base address and length based on the system memory address map, which is configured by the BIOS (or UEFI) at boot. MMIO accesses are forwarded to the PCIe root complex, converted to PCIe packets, and routed to the GPU. Modern GPUs use MMIO BARs to

expose hardware registers for configuring the device and frequently accessed onboard device memory, e.g., command queues for controlling computation.

DMA enables the GPU hardware to directly access CPU memory without the help of the CPU, using address translation through a host-configured IOMMU to implement memory protection and ensure the device only reads and writes data belonging to the application it is serving. GPUs rely on DMA for bulk data transfer.

The host hypervisor and operating system control the PCIe bus, which routes packets to multiple devices connected to the PCIe root complex in a tree topology. Packets in transit to/from the GPU may be visible to other devices. Privileged host software may change the routing topology dynamically and install pseudo-devices that allow it to sniff traffic. Securing communication with the GPU must defend against these passive and active PCIe attacks.

Applications use GPUs through high-level, vendor-provided APIs such as CUDA [NVIa] and HIP [HIP]; they include a user-level runtime and OS-level driver that communicate through a combination of `ioctl` system calls and memory-mapped command queues. The driver is responsible for creating mappings from virtual memory to physical MMIO regions. After these privileged operations are complete, any software that has a mapping (user or OS) may communicate directly with the device using registers or command queues exposed through the MMIO regions.

While memory management, synchronization, and other features (e.g.,

IPC and power management) require interaction with the driver state (e.g., creating and managing memory mappings), workloads that pre-allocate all of their required GPU memory and use only data transfer and kernel launch primitives can function completely by writing commands into the GPU's command queue. It is possible to construct and submit these commands without referring to any state maintained by either the runtime or the driver.

3.3.2 GPU Trusted Execution Environments

While there are no GPU TEEs available on the market today, Graviton [VVB18] is a detailed proposal from the literature that provides the basic functionality that any GPU TEE (or indeed any TEE) should provide. Graviton provides secrecy for GPU code and input data, integrity for the GPU computation, and remote attestation for the computation's initial state. Graviton achieves most of its functionality by changing the GPU firmware, so it does not require extensive changes to the GPU hardware itself. Modern GPU firmware runs on a fully programmable control processor [NVIc], making Graviton's changes achievable. We explain GPU TEE functionality by saying what the GPU does, but the implementation could be firmware, hardware, or both.

A secure channel ensures the integrity, secrecy, and ordering of the commands sent to the GPU. Before computation begins, the client machine and the GPU agree on a shared symmetric key via a key exchange protocol (e.g., Diffie-Hellman). The client uses this key to send commands using a protocol like transport layer security (TLS), which provides a secure channel.

The GPU assures the integrity of the computation. The GPU attests the initial execution conditions to the remote user, who can verify that the provider initialized the GPU with code and data loaded into the expected address ranges with the expected permissions and that the hardware generating the attestation is genuine. There are many variations on remote attestation, but it is a common feature for modern enclaves like SGX [CD16] and Keystone [LKC+18].

The GPU divides its memory into untrusted and trusted regions (reminiscent of SGX's EPC). The untrusted host OS or Hypervisor can DMA into untrusted GPU memory, enabling efficient data transfers; the GPU can then copy data between untrusted GPU memory and trusted memory. This mechanism provides GPU memory protection even though the IOMMU is under control of the untrusted kernel. Graviton disables unified memory, allowing privileged CPU code to demand page GPU memory and exposes side-channel memory access information.

The GPU TEE should turn off or refuse to report the state of any performance counters. Recent GPU side-channel attacks [NNQAG18, FGBR18] have successfully used timing data from GPU performance counters. Similarly, the GPU TEE should avoid reporting the values of physical sensors such as temperature or power use. These sensors can leak information about execution since the GPU draws different amount of power or generates different amounts of heat depending on the workload. Preventing programmatic access to sensors does not prevent an adversary from measuring these properties externally; to

defend against an adversary with that power would require orthogonal techniques not discussed here.

Chapter 4

Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data

Trusted Execution Environments (TEEs) prevent public cloud providers from directly observing secret data during execution, but they do nothing to prevent code inside the TEE from divulging secrets. While this is not a problem when the data owner controls (or can vet) TEE code, it *is* a problem for the vast majority of public-cloud-hosted applications. Data-processing services like image editing (Pixlr [Pix]), tax preparation (TurboTax [Int]), and personal health analyses (23andMe [23ab]) are often composed of proprietary code controlled and deployed by third parties. Vetting code is not an option for these services since their source code contains secrets that the service providers want to protect.

This chapter is based on the previous publication: "Ryoan: A Distributed Sandbox for Untrusted Computation on Secret data", by Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel in the ACM Transactions on Computer Systems (TOCS), Vol.35, No.4, December 2018 [HZX+18]. That publication is an expanded version of the original work which was published with the same title and authors in the proceedings 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI), Savannah, Georgia, USA, November, 2016 [HZX+16]. My contributions to these publications include designing the Ryoan execution model and contributing to its implementation, porting Native Client to SGX, implementing copy-on-write checkpointing, building the image processing application, and evaluating the performance overheads.

User inputs to data-processing services are often sensitive, such as tax documents and health data, which creates a dilemma for the user. If users want to keep their data secret, they either have to give up using the services or hope that they can be trusted. Without additional mechanisms, there is nothing to prevent service software from leaking data (intentionally or unintentionally) beyond the confines of a TEE.

Companies providing data-processing services for users often wish to outsource part of the computation to third-party cloud services, a practice called "software as a service (SaaS)." SaaS encourages the decomposition of problems into specialized pieces that the service providers can assemble on behalf of a user. For instance, 23andMe might want to combine thier health expertise with Amazon's machine learning expertise and robust cloud infrastructure. However, 23andMe now finds itself a user of Amazon's machine learning service and faces the same dilemma—it must disclose proprietary correlations between health data and various diseases to use Amazon's machine learning service. In these scenarios, the owner of secret data has no control over the data-processing service.

We propose Ryoan[1], a distributed sandbox that allows users to keep their data secret in data-processing services, without trusting the software stack, developers, or administrators of these services. First, Ryoan provides sandbox instances to confine individual data-processing modules and prevent

[1]Ryoan is a sandbox, and its name is inspired by a famous dry landscape Zen garden that stimulates contemplation (Ryōan-ji).

them from leaking data; second, Ryoan uses TEEs to allow a remote user to verify the integrity of the sandbox instances and protect their execution; third, Ryoan allows confined code modules to communicate in controlled ways, enabling flexible delegation among mutually distrustful parties.

Ryoan faces issues beyond those faced by TEE-based shielding systems such as Haven [BPH15]. TEEs protect an application that the user trusts and does not collude with the infrastructure. In Ryoan's threat model, neither the application nor the infrastructure is under the user's control. The application and the infrastructure may try to steal the user's secrets by colluding via *covert channels*—even if the application itself is isolated from the provider's infrastructure using enclave protection. Ryoan's goal is to prevent such covert channels and stop an untrusted application from intentionally and covertly using users' data to modulate events like system call arguments or I/O patterns, which are visible to the infrastructure.

Ryoan confines untrusted modules that make up an untrusted application. Confining untrusted code is a longstanding problem that remains technically challenging [Lam73]. Ryoan meets the challenges of confinement by taking advantage of hardware-supported enclave protection and assuming a request-oriented data model. Confined modules only process input once and can neither read nor write persistent storage after receiving the input. This model limits Ryoan's applicability to request-oriented server applications—but such servers are the most common way to bring scalable, data-processing services to large numbers of users.

Ryoan uses multiple instances of a trusted sandbox to confine an application. We based the trusted sandbox used in the Ryoan prototype on Native Client (NaCl) [YSD+09, SMB+10], a state-of-the-art, user-level sandbox (it can be built as a standalone binary, independent from the browser). NaCl uses compiler-based techniques to confine untrusted code rather than relying on address space separation, a property necessary to be compatible with SGX enclaves[2]. The Ryoan sandbox safeguards secrets by controlling explicit I/O channels, and covert channels such as system call traces and data sizes.

The Ryoan prototype uses SGX to provide hardware enclaves. Each SGX enclave contains a sandbox instance that loads and executes untrusted modules. The sandbox instances communicate with each other to form a distributed sandbox that enforces strong privacy guarantees for all participating parties—the users and different service providers. Ryoan provides taint labels (similar to secrecy labels from DIFC [ML97]) that users and service providers define, allowing them to ensure that Ryoan confines any module that processes their secrets.

Contributions. Ryoan's security goal is simple: prevent leakage of secret data. However, confining services over which the user has no control is challenging without a centralized trusted platform. We make the following contributions:

[2]"Enclave" is what Intel calls an SGX TEE

- A new execution model that allows mutually distrustful parties to process sensitive data in a distributed fashion on untrusted infrastructure.

- The design and implementation of a prototype distributed sandbox that confines untrusted code modules (possibly on different machines) and enforces I/O policies that prevent leakage of secrets.

- Several case studies of real-world application scenarios to demonstrate how they benefit from the secrecy guarantees of Ryoan, including an image processing system, an email spam/virus filter, a personal health analysis tool, and a machine translator.

- Evaluation of our prototype's performance characteristics by measuring the execution overheads of each of its building blocks: the SGX enclave, confinement, and checkpoint/rollback. The evaluation is based on both SGX hardware and simulation.

Application limitations. Ryoan forces applications to adopt a request-oriented data model. This data model is sufficient for batch processing of mostly unique inputs. There are application behaviors that do not map cleanly—or at all—onto Ryoan's data model. Below are classes of application behavior that Ryoan does not support.

Storage. Ryoan is not suited for storage; it is intended to safeguard computation on sensitive inputs. Once a Ryoan module has seen user data, Ryoan prevents the module from writing to persistent storage.

Network metadata. Ryoan takes no steps to protect network connection metadata like the user's IP address or the length of packets. Ryoan protects user data but does not protect connection metadata (though systems exist that protect connection metadata, e.g., Tor [DMS04] hides a client's network address from the server).

Repeated computations on the same/similar input data. Ryoan cannot eliminate all timing channels, but it does mitigate their effects with its request-oriented data model. For services that repeatedly process the same or very similar inputs, Ryoan might leak too much confidential information. For example, some online photo services intend for users to repeatedly read and edit photos. Ryoan is not well suited for these services because with enough repeated input, untrusted modules can exfiltrate the input data.

Multi-user computation. If a single request contains secrets from multiple, mutually distrusting users, Ryoan cannot isolate them. Ryoan tracks data flows at a request granularity, and applications are free to mix data within a single request, even if that data comes from different, mutually distrusting users.

4.1 Ryoan's speciation of the malicious public cloud threat model

We consider multiple, mutually distrustful parties involved in data-processing services. A service provider is not trusted by the users of the service to keep data secret; if the service provider outsources part of the computation to other service providers, it becomes a user of those service providers and does not trust them to provide secrecy either. Each service provider can deploy its software on its computational platform, or use a third-party cloud platform that no service provider trusts. We assume that users and providers trust their code and platform, but do not trust each other's code or platforms. Everyone must trust Ryoan and SGX.

A service provider might be the same as its computational platform provider, and the two might collude to steal secrets from their input data. Besides directly communicating data, the untrusted code may use covert channels via *software interfaces*, such as system call sequences and arguments, to communicate bits from the user's input to the platform.

Ryoan takes no steps to prevent each party from leaking its own secrets intentionally or via bugs. This model is suited for the case where the service provider deploys code on its own computational platform (see section 4.3.3 for more discussion). When executing on the platform of another provider, Ryoan provides protections against a malicious OS. For instance, Ryoan validates system calls to prevent Iago attacks [CS13] (similar to Haven [BPH15], Inktag [HKD+13], Sego [KDL+16], SCONE [ATG+16], and Graphene-SGX [TPV17]),

and encrypts communication to protect data secrecy. Application designers may use orthogonal techniques [RLT15, CVDBDS09, CDE08, ZWC+13, KMPS11] to mitigate the unintentional disclosure of application secrets. Similarly, we assume computational platform providers are responsible for protecting their own secrets (e.g., the administrator's password).

Although we consider covert channels based on software interfaces like system calls, we do not consider side or covert channels based on *hardware limitations* (§3.2.2) or *execution time*. Untrusted enclaves can leak bits by modulating their cache accesses, page accesses, execution time, etc. Such channels are themselves technically difficult and often require dedicated systems to address adequately [LCW13, ZAM12, KPMR12, CLD16, FWZ+16]. Many well-regarded secure system designs factor-out side/covert channels based on hardware limitations or execution time, at least to some degree [VEK+07, ZBwKM06, LGV+09, PBR+14, BPH15], because doing so enables progress in designing and building secure systems. While we do not claim to prevent the execution-time channel, Ryoan does limit the use of this channel to once per request (see section 4.3.1 for a more robust explanation).

4.2 Native Client background

Google Native Client (NaCl) [YSD+09, SMB+10] is a sandbox for running Arm/x86/x86-64 native code (a NaCl module) using software fault isolation. NaCl consists of a verifier and a service runtime. Application code is compiled by a specialized compiler that lays out instructions in a way that can

be easily validated by the verifier, so the compiler need not be trusted. The verifier disassembles the binary and validates the disassembled instructions as being safe to execute, to guarantee that the untrusted module cannot break out of NaCl's SFI sandbox.

NaCl executes system calls on behalf of the loaded application. System calls in the application transfer control to the NaCl runtime, which determines the proper action. Ryoan cannot allow the application to use its system calls to pass information to the underlying operating system. For example, if Ryoan passed read system calls from the application directly to the platform, the application could use the size and number of the calls to encode information about the secret data it is processing. We discuss the details of the confinement provided by Ryoan in Section 4.3.5.

Rowhammer attacks. Attackers can use rowhammer attacks from confined code to break older versions of the NaCl sandbox [KDK+14]. In a rowhammer attack, the adversary forces the processor to write a cache line back to memory rapidly, flipping bits in otherwise unwriteable memory locations. Modifying normally protected memory allows untrusted code to break NaCl's sandbox by violating its invariants. Newer versions of NaCl disallow `CLFLUSH` instructions, a core mechanism used in the original rowhammer attack [KDK+14]. There have been successful rowhammer attacks against NaCl using non-temporal stores [QS16] (and NaCl forbade non-temporal stores in response).

Regardless of NaCl's vulnerability to rowhammer attacks, any rowham-

mer attack mounted against an SGX enclave would cause memory integrity checks to fail when an affected cache line was read [JLLK17]. Thus any rowhammer attack mounted by code inside our outside the enclave becomes a denial-of-service attack (out of scope).

Spectre and meltdown attacks. Native Client's sandboxing mechanisms prevent sandboxed code from mounting the Meltdown [LSG+18] attack. Meltdown requires the attacker to issue memory operations for kernel addresses, which become arguments to speculatively executed instructions; Native Client restricts the memory addresses that untrusted code can only reference a 4GB range, which never overlaps with kernel addresses.

Native Client's sandboxing mechanisms also make it more challenging to mount Spectre attacks. Spectre attacks rely on the attacker's ability to train the hardware branch predictor. The attacker trains the branch predictor to make wrong predictions during the execution of the victim. Confined Native Client code is always position-independent, and the targets of its indirect branches must be aligned blocks within a 4GB range. These measures reduce the attacker's freedom in manipulating the branch predictor, thereby shrinking the attack surface.

4.3 Design

Ryoan is a distributed sandbox that executes a directed acyclic graph (DAG) of communicating, untrusted modules which operate on secret data.

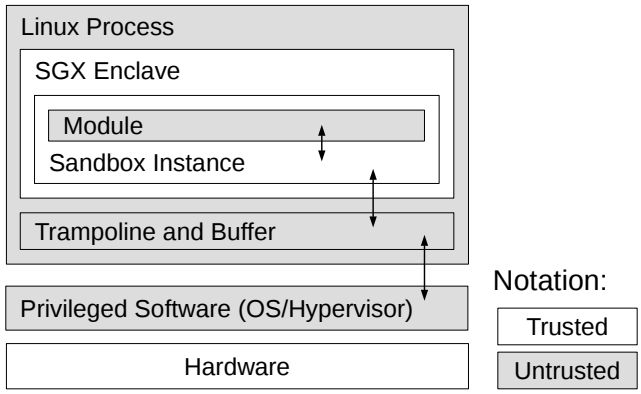

Figure 4.1: One of several sandbox instances that make up a Ryoan deployment. The privileged software includes an operating system and an optional hypervisor.

Ryoan's primary job is to prevent the modules from communicating any secret data outside the confines of the system (including external hosts and the platform's privileged software).

A module consists of code, initialized data, and the maximum size of dynamically allocated memory. For backward compatibility, Ryoan modules support programs written for `libc`, including fully compiled languages and runtimes built on top of `libc`. A Ryoan module can be a Linux program, or it could contain a library operating system [BPH15]. SGX disallows ring 0 execution in enclaves, so Ryoan cannot directly support an operating system or hypervisor.

Confining modules without trusting privileged software (i.e., the operating system and hypervisor) is Ryoan's chief technical challenge. In the worst case, the modules and privileged software can collude to steal secrets.

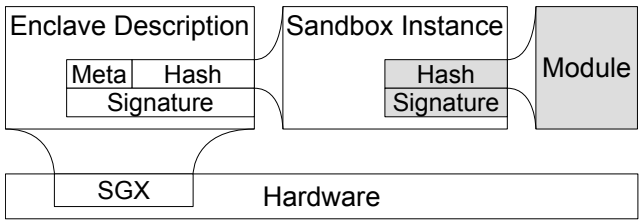

Figure 4.2: The Ryoan chain of trust. SGX hardware attests that a valid sandbox instance is executing (Hash) with an intended SGX configuration (Meta). The sandbox instance ensures that it loaded the expected binary with a signed hash from the software provider (gray).

The possibility of collusion forces Ryoan to consider any behavior visible to privileged software (henceforth *externally visible behavior*) to be a potential channel for leaking secrets.

Figure 4.1 shows a single instance of the distributed sandbox. A principal (e.g., a company providing software as a service) can contribute a module that Ryoan loads and confines, enabling the module to operate on secret data safely. We will refer to any principle that provides a module as a *module provider*. The NaCl sandbox uses a load-time code validator to ensure that the module cannot violate the sandbox by accessing memory outside its address range or making system calls without Ryoan intervention.

Ryoan assures its secrecy and integrity by executing in hardware-protected enclaves provided by SGX. Hardware attests to Ryoan's initial state, and in doing so, hardware becomes the anchor for Ryoan's chain of trust (Figure 4.2). SGX generates an unforgeable remote attestation for the user that a sandbox instance executes in an enclave on the platform. The user can establish an encrypted channel that they know terminates within that sandbox instance.

SGX guarantees the enclave cryptographic secrecy and integrity against manipulation by privileged software.

A master enclave creates all sandbox instances, and they establish cryptographically protected communication channels among themselves as specified by the user. Once the providers have instantiated the modules, the master forwards attestations for each module to the user. The user verifies that the configuration matches their specifications. Then the user inputs their secret data. Ryoan provides simple labels to protect secret data added by modules in the DAG (§4.3.3). All Ryoan's sandbox instances form a distributed sandbox that protects secret input data from being leaked by the untrusted modules that operate on it.

Ryoan prevents modules from leaking sensitive data by decoupling externally visible behaviors from the content of secret data. Anything the module does in response to input data is in danger of being a side channel that communicates it. Ryoan, therefore, makes the module's externally visible behavior independent of the input data. SGX hardware limits externally visible behaviors to explicit stores to unprotected memory and the use of system services (system calls). The NaCl toolchain and runtime eliminate unprotected stores.

Ryoan eliminates most system calls by providing their functionality from within NaCl. For example, Ryoan provides `mmap` functionality by managing a fixed-sized memory pool within the SGX enclave. However, untrusted modules must read input and write output, so Ryoan provides a restricted I/O model that prevents data leaks (e.g., the output size is a fixed function of

Module property	Enforced by	Reason
OS cannot access module memory (§3.2.1).	SGX	Security
Initial module code/data verified (§3.2.1).	SGX	Security
Can only address module memory (§4.2).	NaCl	Security
Ryoan intercepts syscalls (§4.2,§4.3.1).	NaCl	Security
Cannot modify SGX state (§4.3.2).	NaCl	Security
User defines topology (§4.3.2).	Ryoan	Security
Data flow tracked by labels (§4.3.3).	Ryoan	Security
Memory cleaned between requests (§4.3.1).	Ryoan	Security
Module defines initialized state (§4.3.4).	Ryoan	Performance
Unconfined initialization (§4.3.2).	Ryoan	Compatibility
In-memory POSIX API (§4.3.5).	Ryoan	Compatibility

Table 4.1: Properties Ryoan imposes on untrusted modules, the technology that enforces them, and the reason Ryoan imposes them.

input size). Table 4.1 summarizes the properties Ryoan imposes on modules to achieve secure decoupling of observable behavior from secret input data.

Figure 4.4 shows an example of Ryoan processing input from user Alice whose sensitive data is processed by both 23andMe and Amazon. Each sandbox instance executes in an enclave on the same or different machines. The host machine(s) might be provided by 23andMe, Amazon, or a third party. In all cases, Ryoan assures no leakage of the user's secrets and prevents leakage of any trade secrets used by 23andMe and Amazon.

4.3.1 Restricted I/O model

In most cases, Ryoan disallows access to or replaces system services to eliminate module-controlled externally visible behaviors. However, Ryoan cannot replace I/O, so it must be allowed in some form (since Ryoan does not

control devices directly). Instead of replacing it, Ryoan enforces a restricted I/O model upon modules. The I/O model ensures that data flow is always independent of the input data; Ryoan never moves data in response to requests of the untrusted module once the module has read its input data. This safety property is sometimes called data obliviousness [OSF+16].

Ryoan requires modules to be request oriented: input can be any size, but each input is an application-defined "unit of work." For example, a unit of work can be an email when classifying spam or a complete file when scanning for viruses. Each module gets a single opportunity to process a single unit of work. After generating output, the module must be destroyed (or reset, see § 4.3.4) to prevent it from sending the secrets of one user to another, or using the processing time of future requests to leak information about past requests (see § 4.3.1 for a full discussion).

Units of work can be any size, but Ryoan ensures that data flow patterns do not leak secrets from input data by making module output size a fixed, application-defined function of the input size. Ryoan protects communication with the following rules: (1) Each sandbox instance reads its entire input from every input-connected sandbox instance before the module starts processing. (2) The size of the output is a fixed function of the input size, specified as part of the DAG. Sandbox instances pad or truncate all outputs to the exact length determined by the function. (3) Each sandbox instance is notified by its module when its output is complete, and it writes the module's output to all output-connected sandbox instances. Sandbox instances encapsulate module

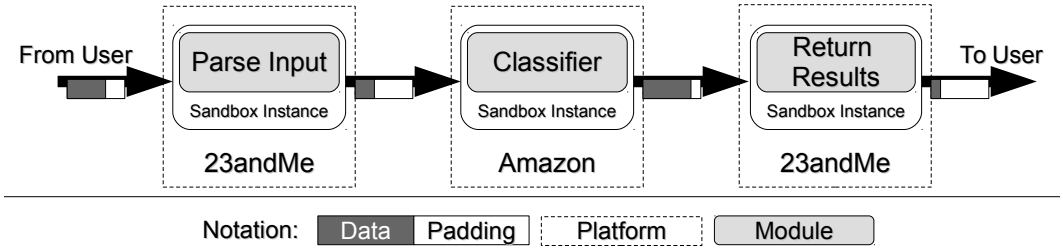

Figure 4.3: Ryoan's distributed sandbox. Modules contributed by principals– in this case, the platform providers, 23andMe and Amazon–are confined to process users' data safely.

output in a message that contains metadata that describes what part of the message is module output and what part is padding (if any). Receiving sandbox instances interpret metadata and remove any padding before exposing the data to its module. These rules are sufficient because they ensure that output traffic is independent of input data (though there are possible alternatives, for example, each request could specify its output size).

Consider the scenario in Figure 4.3. Each input comes from a user. The user can choose to leak the input size or hide it by padding the input. The description of the application specifies that (1) Ryoan pads the output of 23andMe's first module to a fixed size defined by 23andMe which can hold the largest possible user input, (2) the output of Amazon Machine Learning's classifier module is padded to a fixed size to encode the classification result, and (3) Ryoan also pads the response to the user from 23andMe's second module, this time to a fixed size that can hold the largest possible result. Each sandbox instance must receive the complete input of a work unit before executing its module.

Ryoan ensures that output size is a fixed function of the input, so it is a module's mistake if it is not large enough. Ryoan will truncate outputs that are too large and pad outputs that are too small. However, a module author should be able to describe the maximum possible output for a given input-request size. For example, a spam detector's output will be the input mail message plus a constant size sufficient to hold the spam rating for the email. For many tasks, it is easy to bound the size of the output based on the input. For example, it is straightforward to bound the size of a machine learning model with a known topology for a known task with known training data, or to bound the size of a translation from one human language to another.

Processing-time channels. While Ryoan carefully controls I/O, the module controls the amount of time it takes to carry out its computation. A module could use this fact to construct a timing channel by varying the length of time it takes to generate an output based on secrets in the input data. Ryoan takes the following steps to limit leaks through processing time channels:

- *One shot at input data.* Ryoan allows each module to process its input data exactly once, with no opportunity to carry forward state from one input to the next. This one-shot policy limits data leakage. Ryoan enforces the one-shot policy by (1) requiring that the data processing topology be a DAG to avoid cycles; (2) disallowing access to any state modified by processing a different unit of work; (3) preventing input replay attacks by re-initializing all secure connections if any connection is ever broken. Secure communication protocols

contain protection against replay attacks [YL08], so the re-initializing of broken links prevents input replay. Note that the OS can pause or stop the execution of an SGX enclave, but it cannot roll back its state [Int14], which means the cloud provider cannot roll back the state of a secure connection. Ryoan itself uses high-quality randomness available via the processor's RDRAND instruction to establish secure connections, which does not rely on the OS.

- *Randomness.* Users can specify whether confined modules need access to randomness. If the user allows, a module can access randomness via the processor, e.g., Intel's RDRAND instruction. Ryoan does not allow confined modules to get randomness from the operating system. Access to randomness means a malicious module can leak random bits from an input, for example, by choosing an input bit at random and leaking it using its processing time. If the user repeats input data, a malicious module with access to randomness can eventually leak the entire input over its processing-time channel, even though it only leaks once for each input unit of work. Using a fixed processing time eliminates this channel.

Some natural types of input data can function as a source of randomness. If a computation's input contains ever-changing metadata (e.g., an embedded timestamp of the request), then a confined module can use these changing bits to seed a pseudo-random number generator and leak multiple bits from the semantically identical input. Just like users must take care to prevent leaking the size of their input data, they must also take care to avoid semantically identical inputs encoded into different bit representations.

Below are other design choices that would provide stronger leak mitigation. They are not part of the prototype.

- *Fixed processing time.* Timing channels can be eliminated by forcing a fixed processing time whose length is determined before the module has seen any data. The OS cannot directly determine when the module completes, and thus the Ryoan runtime can pad execution time by busy waiting. However, controlling its timing without the cooperation of the operating system is a challenge. Fixed processing time can be quite expensive for computations with widely variable run times because all run times would be padded to the worst case. However, fixed processing time can be quite modest for computations with highly predictable run times (e.g., evaluating certain machine learning models like decision trees) or light throughput requirements. Fixed-time execution does not leak information, though we defer to future work building a sandbox instance that supports it. Execution time could also be a fixed-function of input length, to add flexibility with no loss of security.

- *Quantized processing time.* Reducing the granularity of potential processing times helps to mitigate processing time channels. Systems do this by padding execution to a fixed number of quantized, pre-defined values [TLW+09, ZAM11, AZM10, ZAM12]. Because Ryoan only allows modules to see sensitive data once, enforcing quantized execution would limit the amount of data individual modules can leak to the logarithm of the number allowed execution durations. For instance, if the code terminates after one of eight different statically determined intervals, it leaks three bits.

4.3.2 Secure initialization

Ryoan's secure initialization ensures that modules are loaded correctly by genuine sandbox instances in the specified topology for a particular application. A Ryoan application is described by a *DAG specification*, which specifies how modules should be connected (always a DAG for safety, see § 4.3.1). The user either defines the DAG specification or explicitly approves it.

Initializing the application. A bootstrap enclave (which we will call the *master*) receives the DAG specification to start initialization. Upon receiving the DAG specification, the master requests that the platform instantiate enclaves that contain sandbox instances for modules listed in the specification. Different machines or even different providers can host these enclaves. The master uses attestation to verify each sandbox instance's validity, then informs the sandbox instances of the location of their neighbors in the DAG specification. Sandbox instances establish cryptographically protected communication channels via key exchange with their neighbors using the appropriate untrusted communication medium (e.g., the network or local inter-process communication) as transport.

The user can verify the master's validity via attestation and ask whether the provider has initialized the desired topology. If this is true, the user establishes secure channels with the entry and exit sandbox instances of the DAG, and data processing begins.

The master is convenient but not essential to our design. The only

requirement is that the user receives some statement, attested to by hardware, that the cloud provider did not misbehave. For instance, instead, the sandbox instances themselves could act as a kind of decentralized master and forward attestations of their neighbors to the user.

Ryoan identity and module identity. SGX attests to the sandbox instances using processor hardware, and the sandbox instances, in turn, attest to the modules' initial state using software cryptography (Figure 4.2). SGX supports two forms of identity, one based on a hash of the enclave's initial state (MRENCLAVE) and one based on a public key, product identifier, and security version number (MRSIGNER). SGX can verify Ryoan using either form of identity; our prototype uses MRENCLAVE. Ryoan can support software analogs of either identity for untrusted modules; the prototype identifies modules by the public key that signs them.

Module initialization. A sandbox instance begins by verifying that its module matches the DAG specification. Upon successful verification, the sandbox instance continues by loading and validating its module. Successfully validated modules are allowed to initialize. While initializing, the module is not confined and has full access to the system services exposed by vanilla NaCl. Non-confined initialization makes module creation more efficient, and it makes porting easier because the initialization code can remain unchanged. Modules signal Ryoan when initialization is complete by calling `wait_for_work`,

a routine implemented by Ryoan. Once a module is initialized, it processes a request, generates its output, and then is destroyed or reset to prevent the accumulation of secret data.

Ryoan module validation ensures that modules are safe to execute by enforcing a set of constraints on the loaded code. Ryoan uses NaCl's load-time code validator to ensure that the module's code adheres to a strict format. NaCl's code format is designed to be efficiently verified and efficiently sandboxed, restricting control flow targets and cleanly separating code from data. Memory accesses are confined to remain within the address space occupied by the module, including execution fetches. The detailed guarantees of NaCl are available as prior work [YSD+09, SMB+10], and Ryoan does not change the base guarantees of the NaCl sandbox. Ryoan adds the constraints that modules may not contain any SGX instructions, and that control flow is constrained to the initial module code, i.e., Ryoan disallows dynamic code generation.

Sandbox instance migration. To balance server utilization, Ryoan might periodically reconfigure the deployment of the data processing DAG. Because Ryoan processes secret data once, it does not maintain or migrate any persistent state. However, modules might maintain persistent data, for example, databases for initialization. Ryoan makes no guarantees about a module's persistent state; module providers should consider their trust relationship with the platform provider before depending on the fidelity of any state stored by the

platform. If a module stores persistent data, then the service provider is expected to make that data available to the module when it is re-initialized after migration, e.g., storing it in a distributed data store accessible on the new node.

The Ryoan prototype only supports the most coarse-grained migration achieved by shutting down the processing DAG and recreating it on a new set of nodes. Should migration become a frequent operation, the maser enclave could coordinate migration as an optimization, e.g., it could migrate only certain nodes in the DAG.

4.3.3 Protecting module provider secrets

Ryoan uses security labels to prevent module provider secrets from flowing back to the user. Conceptually, a label is a set of *tags*, where each tag is an opaque identifier drawn from a vast universe that identifies a principal, indicating secrets from this principal. Ryoan uses public keys as tags. Ryoan assigns the user's tag to any data provided by the user. Module binaries are signed; a loaded module's tag is the public key, which correctly verifies the signature on its binary. A module provider could use different key pairs to sign its module binaries, enabling privilege separation.

Ryoan adapts previous label-based systems to enable multiple mutually distrustful modules to process sensitive data cooperatively. Ryoan labels are similar to labels in DIFC systems [ML97, VEK+07, PBR+14, ZBwKM06, KYB+07, LGV+09, PBR+14], but are far simpler. Ryoan labels are

only used to reason about data secrecy (not integrity), and are coarse-grained; Ryoan applies labels to entire modules and the data they generate. Ryoan's use of labels could also be thought of as taint tracking [CPG+04] at enclave-level granularity, with per-principal taint classes. Taint is attached to data at the unit of work granularity (where the units of work are application-defined).

Label manipulation rules. Each module is created with an empty label, and can add or remove a single tag that corresponds to its principal — each module can declassify its own secrets. When a module reads data with a non-empty label (e.g., from a user or another module's output), the module's label is replaced with the union of the data's label and the module's old label. Ryoan marks a module's output data with the module's label.

In Figure 4.4, Alice's input is labeled with their tag, and the first 23andMe module adds the 23andMe tag, to make sure that its secrets cannot flow back to the user after handing them off to Amazon's machine learning module. This control is essential since the user is in control of the topology. The second 23andMe module removes its tag from its output's data label.

In a sense, 23andMe's public key creates a group, and both modules are members of the group—verified by Ryoan because 23andMe signed both modules with that key. Ryoan is trusted to remove the user's tag when it communicates over a protected and authenticated connection to the user.

Non-confining labels. If a module's label does not contain tags from other principals, the module is not confined. Such labels are called *non-confining labels*. A module with a non-confining label may perform any file system operation, network communication, or address space modification permitted by Ryoan and NaCl. For example, it can freely initialize its state by reading from the network or file system. Ryoan allows unfettered access to external resources because the principal's tag means that the module may have seen secrets only from itself. In Ryoan's threat model, each principal trusts their module not to leak their secrets (§4.1) and to validate any data it receives from an untrustworthy source.

In many DIFC systems, principals are independent of the application code, e.g., multiple users (principals) use the same wiki Web application, and the users do not trust the application [VEK+07,PBR+14,ZBwKM06,KYB+07, LGV+09]. Ryoan allows application owners (module providers) to be principals who trust their own code, which is different from the standard DIFC model. Although a module provider's code may have bugs that cause it to release its own secrets in its output, that is not within the threat model for Ryoan and can be mitigated using orthogonal techniques (§4.1). Ryoan protects a principal's data when it is processed by modules that are not under the principal's control.

A module provider can host its modules and secret data on its machines to protect them. However, if it chooses to use a third-party computational platform that it does not trust, its modules containing non-confining labels

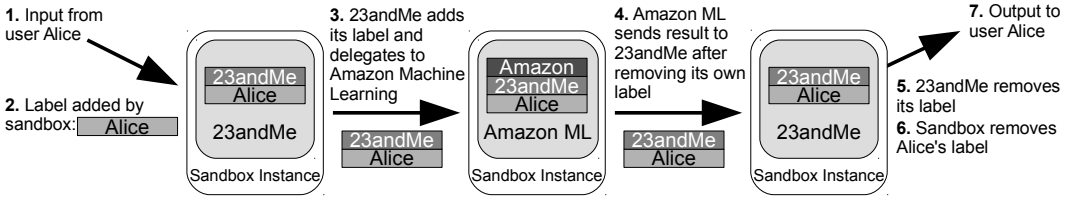

Figure 4.4: Sandbox instances manage labels on data and modules. The user's tag is propagated to all modules, making them confined after receiving input; for example, Ryoan keeps 23andMe's tag when it outsources to Amazon Machine Learning to prevent leaking 23andMe's secrets.

need encryption to protect persistent secrets from the platform. Ryoan uses the SGX sealing feature to store secret data on behalf of modules. Sealing provides an encryption key only accessible to enclaves with the same identity executing on the same processor. For Ryoan, all enclaves contain sandbox instances and have the same identity. The module passes any data that it wants to persist securely to Ryoan, which adds metadata, including the module's public key. Ryoan seals the data and metadata and writes the result into a file. The metadata allows Ryoan to persist data on behalf of different modules and allows it to restrict any module's access to its data.

Confining labels. When a module's label contains tags of other principals (as a result of receiving secrets from a user or another module's output), Ryoan confines it. We call such labels *confining labels*. A confining label indicates the module may have seen the secrets of other principals; Ryoan must prevent the module from leaking those secrets.

Ryoan prevents modules with confining labels from persisting data.

As a result, Ryoan's label system is far simpler than DIFC systems [VEK+07, ZBwKM06,PBR+14,KYB+07,ML97]. Confined modules have seen secret data from other principals, so allowing them persistent storage violates Ryoan's "one-shot" request-oriented data model—a module processes a request once and only once.

4.3.4 Optimizing module reset

The restrictions necessary to confine modules create execution time and memory space overheads. In this section, we discuss strategies for mitigating these overheads.

Checkpoint-based enclave reset. Creating and initializing modules often requires far more CPU time than processing a single request (see Section 5.6 for measurements). For instance, loading the data necessary for virus scanning takes 24 seconds, orders of magnitude greater than the ≈0.124 seconds it takes to process a single email. Ryoan manages the module lifecycle efficiently using a checkpoint-based enclave reset.

Creating and initializing a hardware protected enclave is slow (e.g., we measured 30 ms for a small enclave). Compounding the problem is that applications often do not optimize their initialization sequence because they assume that it will not be executed frequently. However, Ryoan does not allow any data from one input unit of work to be carried forward to the next. Each input requires that the computation begins from the same, non-secret state,

making initialization a bottleneck.

Ryoan provides a checkpoint service that rolls back the application to an untainted, but initialized, memory state (Figure 4.5). In our prototype, this state is at the first invocation of `wait_for_work`. Ryoan does not allow an enclave that has seen secret input to be checkpointed, because its data model is request-oriented: modules cannot depend on data from past requests to operate. Checkpointing a module that has seen secret data would (potentially) give that module multiple execution opportunities on a single request's unit of work.

Checkpoint restore allows Ryoan to save the cost of tearing down and rebuilding the SGX enclave, and it saves the cost of executing the application's initialization code. Ryoan takes checkpoints once but restores the checkpoint after each request is processed. Therefore, Ryoan makes a full copy of the module's writeable state and simply tracks which pages get modified (avoiding a memory copy during processing); Ryoan only needs restore the contents of modified pages (§4.4.6). SGX provides a way for enclave code to verify page permissions and be reliably notifies the enclave about memory faults, which is necessary to track modified pages.

Batch requests before a reset. A user might want more efficiency by allowing a module to process several input units of work before reset. Whether batching multiple inputs within a single request constitutes a threat is user and application dependent. However, if a module can process more than one

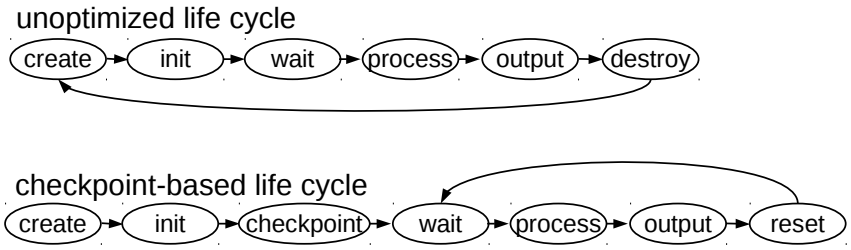

Figure 4.5: Sandbox instance lifecycle: unoptimized vs. checkpoint-based.

unit of work from the same data source, it can accumulate secrets across multiple wait-process-output cycles. Access to more secret data for longer periods exacerbates the problem of slow leaks (e.g., timing channel leaks). For example, an email-filtering module allowed to process multiple emails without resetting could leak multiple bits of a password contained in one email by using the processing-time channel across multiple wait-process-output cycles.

4.3.5 Ryoan's confined environment

Any module with a confining label executes in Ryoan's confined environment. Ryoan's confined environment is intended to prevent information leakage while reducing porting effort. In order to allow code developed for general-purpose computing environments to be used within Ryoan, the trusted Ryoan runtime can provide backward compatibility services. When a module receives the secret data contained within a request, it enters the confined environment and loses the ability to communicate with the untrusted OS via any system call. Therefore, Ryoan provides a system API sufficient for most legacy programs to perform their function without modification. Ryoan pro-

vides these services:

• The most important service is an *in-memory virtual file system*. First, Ryoan allows users to preload files into module memory. The list of preloaded files must be determined before the module is confined, e.g., they can be listed in the DAG specification, or requested by the module during initialization. Ryoan presents POSIX-compatible APIs to access preloaded files that are available even after the module is confined. Second, a confined module can create temporary files and directories (Ryoan keeps them in enclave memory). When the module is destroyed or reset, Ryoan destroys all temporary files and directories and reverts all changes to preloaded files.

• `mmap` calls are essential to satisfy dynamic memory allocation, so Ryoan supports anonymous memory mappings by returning addresses from a pre-allocated memory region. The module must decide the maximum size of that region before it becomes confined.

Ryoan's confined environment is sufficient for many data-processing tasks. For example, ClamAV–a popular virus scanning tool–loads the entire virus database during initialization; when scanning the input such as a PDF file, it creates temporary files to store objects extracted from the PDF. Ryoan's in-memory file system satisfies these requirements.

However, if an application needs a large database that does not fit in memory when processing data, Ryoan cannot support it as a single module. A workaround would be to partition the database and use multiple modules

to load different partitions and perform different parts of the task if that is feasible for a particular application.

Any design alternative that allows access to persistent files (as opposed to Ryoan's in-memory files) must cope with the covert channel created by allowing the OS to see file reads, which might occur based on the computation within the untrusted module. Ryoan eliminates this channel by executing from memory only. All Ryoan modules must fit into memory for their entire execution because any "swapping" done by Ryoan will create a covert channel between the module and the operating system. File access techniques based on oblivious RAM (ORAM [RFK+15,LHH+15]) can hide data access patterns, but at a performance and resource cost that we deem too high.

4.3.6 Protecting Ryoan from privileged software

A sandbox instance requires services provided by the untrusted operating system and possibly the hypervisor. The sandbox instance must check the results coming from the untrusted operating system to make sure it is not misbehaving. We inserted most of these checks into `libc`, which communicates with the operating system. Ryoan-libc is Ryoan's replacement for `libc`, and it manages system call arguments and checks their return values. The Ryoan sandbox code invokes Ryoan-libc through standard `libc` functions, such as the wrappers for system calls (e.g., `read`). SCONE [ATG+16] and Graphene-SGX [TPV17], also modify `libc`.

Iago attacks. Ryoan-libc guards against all known Iago attacks [CS13] by keeping state in enclave memory and carefully checking the results of system calls, e.g., making sure that addresses returned from `mmap` do not overlap with previously allocated memory (like the stack). For Linux, the system call interface can be secured, e.g., by maintaining semaphore counts in enclave memory and duplicating futex [FRK02] memory inside and outside the enclave. Ryoan shares the need for this checking with all systems distrustful of the operating system [HKD+13, KDL+16, CGL+08], though some check at a lower level than system calls [BPH15].

Controlling an enclave's address space. SGX provides user control of memory mapping, including permissions. Ryoan-libc maintains a data structure equivalent to the kernel's list of virtual memory areas (VMAs). It knows about each mapped region and its permissions. Map requests are fulfilled eagerly and verified by Ryoan-libc at the time of the request (i.e., part of the `mmap` call), not at page fault time.

SGX dictates a very specific procedure for verifying enclave mappings. A typical new mapping proceeds as follows: (1) untrusted code notifies the kernel of a new desired mapping via a system call made by Ryoan-libc; (2) the OS selects new enclave page frames to satisfy the mapping and modifies the page tables to map the frames at the requested virtual address with the requested permissions; (3) untrusted user code resumes and passes control to enclave code; (4) enclave code verifies that the mapping completed as expected by

invoking the SGX instruction `EACCEPT` on every new page. The `EACCEPT` instruction accepts a virtual address and protection bits and verifies that the current address space maps that page to a valid, SGX protected 4KB physical frame. New pages added to the enclave always start with read and write permissions, and hardware zeros their contents.

If the user wants something other than read and write permission, SGX provides the `EMODPE` instruction to make them more permissive and the `EMODPR` instruction, which makes them less permissive. `EMODPE` is only available to enclave code while `EMODPR` is only available to privileged software (ring 0, outside of the enclave). If an enclave desires less permissive page access rights, it must signal privileged software to request the restriction. However, it can validate that it was done correctly through another use of the `EACCEPT` instruction.

Ryoan-libc emulates `mmap` behavior by doing work required by SGX on behalf of the user. For instance, if the user expects new pages to have particular contents (e.g., the user privately mapped a file) and to be read-only, Ryoan-libc copies the file into enclave memory and ensures those pages have read-only permissions before returning.

Rollback. Privileged software can rollback any persistent state. Ryoan does not depend on any persistent state, preventing rollback attacks by design. Ryoan also provides mechanisms that allow module providers to avoid dependence on any persistent state. Ryoan's initialization depends only on its initial in-memory state, which is protected and attested by hardware. All other state

is derived from hardware randomness or provided securely at runtime by the sandbox provider.

A module might use persistent state, for example, during initialization before seeing any user-supplied secrets. In Figure 4.3, Amazon's machine learning classifier might load pre-computed parameters stored in a Ryoan-managed per-application directory in the local file system. Module providers should employ encryption, hashing, and rollback protection appropriate to their trust relationship with the platform provider (and with any provider of information).

Persistent state protection for modules is the module provider's responsibility, just as module functionality/correctness is the module provider's responsibility. Ryoan guarantees that once a module sees user data, it cannot leak that data; it does not guarantee that modules act according to specifications, e.g., that a module correctly identifies spam.

Enclave indistinguishability. While SGX enables enclaves to attest their integrity to outside parties, nothing prevents the platform from instantiating multiple copies of enclaves. Ryoan prevents the platform from exploiting this fact by establishing secure channels between different enclaves and between enclaves and the user with never-persisted keys, requiring the user or other enclaves to renegotiate a key with each new enclave (tipping them off to the switch).

4.4 Implementation

The sandbox instance prototype is based on NaCl version 2d5bba1 with the last upstream commit on Jan 19, 2016. We leverage NaCl's existing sandboxing guarantees to control the module's access to the platform. NaCl ensures that the module in the sandbox has no direct access to OS services. We ported NaCl for use in SGX with the introduction of the Ryoan-libc layer. NaCl depends on `libc` to interface with the platform. Ryoan-libc makes system calls on behalf of a sandbox instance after checking that the system call is allowed. We modified `eglibc`'s dynamic linker to support loading Ryoan into enclaves, but all modules must be statically linked. We base Ryoan-libc on `eglibc` 2.19, which is compatible with our version of NaCl.

4.4.1 Constraints of current hardware

Ryoan relies on features from Version 2 of the SGX hardware, while only Version 1 is currently available. Version 2 adds the ability to modify enclaves dynamically, i.e., augmenting an executing enclave with new memory and changing protections on existing enclave memory. Ryoan relies on changing memory protections to implement efficient checkpoint recovery. Furthermore, our first-generation SGX-capable machine makes only a limited amount of physical memory available to SGX (128MB on our machine).

4.4.2 Ryoan-libc

Ryoan-libc manages interactions with the untrusted operating system. The OS cannot read enclave memory, so Ryoan-libc marshals system call arguments into the process' untrusted memory and copies back results. Interposition from `libc` is common for applications that do not trust the operating system [CGL+08, HKD+13, KDL+16], while Haven protects a smaller system interface [BPH15].

Fault handling. Signals allow user-level code to be interrupted by the system. The sources of most signals are unreliable when the OS is untrusted, but SGX allows us to get reliable information about memory faults; this allows Ryoan-libc to expose this information to sandbox instances through the normal signal handler registration interface. Ryoan-libc signal support is currently limited to the memory fault signal (SIGSEGV).

After any fault, exception, or interrupt the OS returns control to untrusted trampoline code contained within the process. For memory faults, rather than simply resuming the enclave where it was paused (as in the normal case), our trampoline code enters the enclave. Inside the enclave, it can read reliable information about the fault from SGX and make necessary arrangements to fix it (e.g., change permissions). After handling the fault, the enclave exits, and then our trampoline resumes the enclave at the instruction that caused the memory fault. We cannot protect the trampoline code from the OS. However, it can only enter the enclave using the EENTER instruc-

tion, which will transfer control to our fault-checking entry point, or resume the enclave using the ERESUME instruction, re-executing the instruction that faulted. If the OS tries to resume the enclave without calling the enclave fault handler, the instruction will simply re-fault.

4.4.3 Module address space

x86-64 NaCl allocates an 84 GB region for a NaCl module with 4 GB of module address space flanked above and below by 40 GB of inaccessible guard pages. However, current SGX hardware only allows enclaves with 64 GB of virtual address space. Fortunately, the original x86-64 NaCl design [SMB+10] overestimated the number of guard pages needed to allow for future changes in the architecture. A detailed analysis [nac] indicates we can remain safe by keeping the upper guard region unchanged but reducing the lower region from 40 GB to 4 GB. Therefore, a sandbox instance requires 48 GB of virtual address space, which fits into current SGX hardware.

4.4.4 I/O control

A sandbox instance controls its module's access to files and request buffers when it is confined, preventing the module from leaking data via direct syscalls.

In-memory virtual file system. A confined module cannot access the file system, but Ryoan implements POSIX-compatible APIs for in-memory virtual

files, including preloaded files and temporary files. Ryoan backs in-memory files with a set of 4 KB blocks indexed by a two-level tree structure (similar to a page table). Ryoan allocates the blocks of a file on-demand as the file grows. The maximum size of an in-memory file is 1 GB. Ryoan backs in-memory directories with a hash table, and we use reference counts to track the lifetime of files. This virtual file system supports standard APIs, including `open`, `close`, `read`, `write`, `stat`, `lseek`, `unlink`, `mkdir`, `rmdir`, and `getdents`. When the module writes a preloaded file, the sandbox instance keeps the original file blocks. When the module resets, Ryoan restores preloaded files to their original versions and deletes temporary files.

Input/output buffers. For each unit of work, a module calls `wait_for_work` (a system service implemented by Ryoan). The sandbox instance reads its entire input from all input channels into memory buffers before returning to the module. After processing the work unit, the module writes its output to a buffer, the sandbox instance flushes the buffer to output channels on the next `wait_for_work` call. Before writing out the data the sandbox instance pads or truncates the output to a size calculated using a fixed function of input size according to the DAG specification. The module accesses these buffers via file descriptors using APIs implemented in the virtual file system, just like using regular pipes or sockets.

4.4.5 Key establishment between enclaves

Sandbox instances implement protected channels using an authenticated encryption algorithm (AES-GCM [MV05]) provided by the libsodium [lib] library. Encryption keys are agreed on at runtime using a Diffie-Hellman key exchange. SGX allows enclave code to embed the key parameters in attestations, accelerating a Diffie-Hellman key exchange between enclaves [sgx15]. On our hardware (§5.6), SGX key exchange takes 1.78ms, while OpenSSL takes 1.90ms. Randomness is required for key exchange, and Ryoan uses the x86 instruction RDRAND to obtain it.

4.4.6 Checkpointing confined code

Ryoan uses page permission restriction and fault information to detect module writes. Recall that SGX provides reliable memory page permissions and information about memory faults; Ryoan does not trust the OS (§4.4.2). The entire module is write-protected by the OS when it is confined. Ryoan verifies that the protection was done using `EACCEPT` . As the module writes, the sandbox instance catches permission faults and records the offending page's address before changing the permissions to allow writes and resumes the module. However, updating the permissions in the page table requires ring-0 privilege. The sandbox instance's signal handler first executes outside the enclave and makes an `mprotect` system call to change the page permissions, to avoid an extra enclave exit. Once that process is complete, it enters enclave mode to update SGX page permissions with `EMODPE` (and performs the bookkeeping

mentioned above). With that done, the handler returns and normal execution resumes.

All written pages are restored to their initial value and made unwritable again to reset the enclave. In our prototype, before Ryoan confines an untrusted module for the first time, the sandbox instance creates a checkpoint by copying the module's complete writable memory state. This copy-on-initialize strategy optimizes the case where sandbox instances are created once and then used and reset for many requests. If the copy-on-initialize cost is too high, Ryoan could incrementally create the checkpoint by doing a copy-on-write for each request, gradually accumulating and preserving unmodified versions of any page modified during any execution.

In our prototype, the Ryoan checkpoints when the module blocks on `wait_for_work` and restores the next time the module blocks on `wait_for_work`. This gives module writers clear semantics about what state will not persist across invocations and allows the sandbox instance to purge any secrets kept in registers.

Restoring a checkpoint does incur additional page faults, which could be used as a channel to leak data. We find these additional faults acceptable as even normal page accesses by the module are a channel between module and OS that SGX does not close [XCP15]. Page faults will continue to leak information about enclave execution until future generations of hardware enclaves can service their page faults (§3.2.2), or SGX provides another hardware fix. To make Ryoan execution on current SGX hardware more secure, we could

save/restore all writable regions of the module instead of tracking individual pages using write protection. This strategy is less efficient but does not leak additional per-page information.

4.5 Use cases

This section explains four scenarios where Ryoan provides a previously unattainable level of security for processing sensitive data. For all examples, the sandbox instances could execute on the same platform or different platforms, e.g., the entire computation might execute on a third-party cloud platform like Google Compute Engine, or a provider's module might execute on its own server. Ryoan's security guarantees apply to all scenarios.

4.5.1 Email processing

A company can use Ryoan to outsource email filtering and scanning while keeping email text secret. We consider spam filtering and virus scanning, using popular legacy applications — DSPAM 3.10.2 and ClamAV 0.98.7.

The computation DAG for this service contains four sandbox instances, each confining a data processing module (see Figure 4.6). An email arrives at the entry enclave over a secure channel, which distributes the email text and attachments to the enclaves containing DSPAM and ClamAV, respectively. The virus scanning and spam filtering modules forward their results to a final post-processing enclave, which constructs a response to the user over a secure channel.

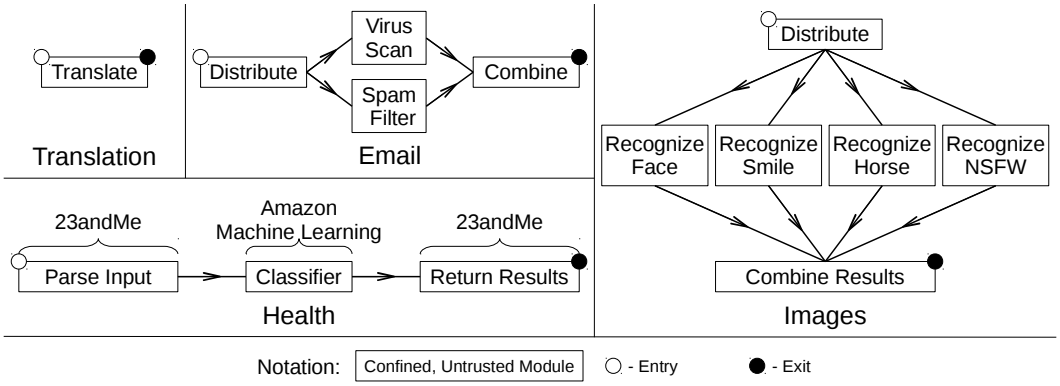

Figure 4.6: Topologies of Ryoan example applications. Nodes in the graph are sandbox instances, though we identify them by their untrusted module. Users establish secure channels with trusted Ryoan code for the source and sink nodes to provide input and get output, respectively.

4.5.2 Personal health analysis

Consider a company (e.g., 23andMe) that provides customized health reports for users based various health data. 23andMe accepts a user's genetic data, medical history, and physical activity log as input; extracts important health features from these data; and predicts the likelihood of certain diseases [23aa]. Since genetic and health information is extremely sensitive, users may not feel comfortable with the company keeping their data. To encourage the use of the service, 23andMe can deploy it with Ryoan, assuring users that the code that processes their data cannot retain or leak their secrets.

23andMe owns its research results about the associations between diseases and health features. However, it may want to use a third-party machine-learning service in the cloud (e.g., Amazon Machine Learning [aml]) to train its model and generate predictions. 23andMe's trade secret is how to map

a user's complex, multi-modal health data onto machine learning features. Amazon Machine Learning provides a way to train models based on unlabeled features and software (a classifier) which queries that model. After training a model this way, 23andMe wants to keep the input to the classifier a secret from parties who have the means to map the inputs back to secret health data: users of their service. Ryoan enables 23andMe to outsource machine learning tasks to Amazon while protecting its proprietary transformation from user data to health features.

Ryoan protects secrecy for both users and 23andMe with the DAG shown in Figures 4.4 and 4.6. 23andMe compiles a training data set which it transmits to Amazon to construct a model. Amazon provides the classifier which queries that model as a Ryoan module. Users provide their genetic information, medical history, and activity log in a request. Upon receiving a user's request, 23andMe's first module constructs a boolean vector of health features and forwards it to Amazon's module. Amazon's module generates predictions based on the model and forwards the result to 23andMe's second enclave, which then forwards the result to the user.

The user's label is kept throughout the entire pipeline so that all the enclaves are confined when receiving the user's input and cannot leak information about the input. Further, 23andMe keeps its label with the request sent to Amazon so that Amazon cannot leak data about 23andMe's health features to other parties (particularly the user) since they cannot remove 23andMe's label in order to release data out of Ryoan's confinement. Amazon's module

passes the results of the classification to another module owned by 23andMe. 23andMe uses this module to verify that its proprietary transformations are not in the output before removing the 23andMe label and sending the results to the user.

Real genetic prediction models are proprietary, unknown to us, and out of scope for this paper; our workload uses general knowledge and best practices. We train a support vector machine (SVM) and choose 20 well-studied diseases and their top 500 correlated genes, according to a database provided by DisGeNet [dis]. We trained the SVM models synthetic data based on that database. Our prototype uses stochastic gradient descent as the training algorithm [Bot], which allows incremental updates to existing models.

4.5.3 Image processing

Image classification as a service is an emerging area that could benefit from Ryoan's security guarantees (e.g., Clarifai [cla] or IBM's Visual Recognition service [ibm]). We envision a scenario where a user wants different image classification services based on their expertise. For example, one service might be known for accurate identification of adult content [MP] while another might do an excellent job of recognizing and segmenting horses. The image processing DAG in Figure 4.6 shows an example where an image filtering service outsources different subtasks to different providers and then combines them. The user's label is propagated to all processing enclaves, causing Ryoan to confine their execution. Our prototype implements all of these detection tasks

using OpenCV 3.1.0. Each detection task loads a model that is specialized in the detection task and would represent a company's competitive advantage.

4.5.4 Translation

A company uses Ryoan to provide a machine translation service while keeping the uploaded text secret. Users upload text to the translation enclave and get the translated text back. Our prototype uses Moses [mos], a statistical machine translation system. We train a phrase-based French to English model using the News Commentary data set released for the 2013 workshop in machine translation [wmt].

4.6 Evaluation

We quantify the time and space costs of Ryoan and its components by measuring the execution of the use cases described in the previous section using a combination of real hardware and emulation.

We measured all benchmarks on a Dell Inspiron 7359 laptop with an Intel Core i5-6200U 2.3 GHz processor (with Skylake microarchitecture and SGX version 1) and 4 GB RAM. We use a laptop because it contains the first SGX-enabled processor we could purchase; however, we validate our measurements using a more recent Intel E3-1270 (see the analysis for SGX Overhead below). We use Intel's SGX Linux Driver [sgxb] and SDK [sgxa] to measure SGX instructions' costs.

To test our implementation and overcome our hardware's limitations,

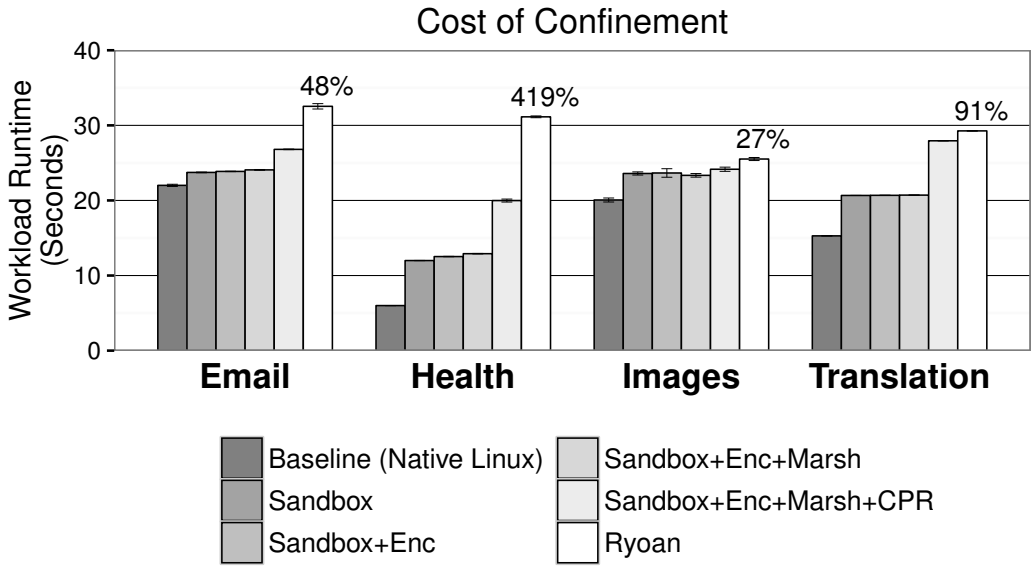

Figure 4.7: Runtimes of applications with Ryoan overheads enumerated. Each bar represents the mean of 5 trials annotated with the 95% confidence interval. Ryoan bars show percent slowdown over native. (Enc: encryption; Marsh: syscall marshaling; CPR: checkpoint restore; Ryoan: Sandbox+Enc+Marsh+CPR+SGX)

Application	Input	
Email	250	emails, 30% with 103KB-12MB attachment
Health	20,000	1.4KB Boolean vectors from different users
Images	12	images, sizes 17KB-613KB
Translate	30	short paragraphs, sizes 25-300B, 4.1KB total

Table 4.2: Inputs for each Ryoan application.

we built an SGX emulator based on QEMU [qem] (full emulation mode), augmented with SGX version 2 instructions. To emulate the performance of SGX V2, we insert delays based on our measurements of current SGX hardware, flush the TLB according to Intel's SGX specification, and estimate overhead for V2 instructions based on the performance of V1 instructions. We could not use OpenSGX [JDK+16], because it lacked 64-bit signals. Our emulator can run a complete software stack, including an SGX-aware Linux kernel.

4.6.1 Understanding workload performance

Figure 4.7 shows a breakdown of the various sources of overheads for Ryoan. The baseline is to run applications built for a native Linux environment and then add sandboxing, encryption, syscall marshaling, checkpoint restore, and SGX (where SGX overheads are a mix of emulation and measurements, see the discussion below). Table 4.2 shows the inputs for each of the workloads, and detailed measurements for each module in the DAG and counts of important events (see Section 4.5 for more details about the workloads).

		Load Size(MB)	Inited Size(MB)	CPR Size	Init Time(s)	CPU Time(s)
Email	Distribute	18.0	18.1	11.6MB	0.59	1.32
	DSPAM	19.6	273.5	45.3MB	11.15	22.10
	ClamAV	21.1	403.9	83.3MB	24.96	29.17
	Combine	18.0	18.1	16KB	0.59	0.11
Health	LoadModel	19.3	19.4	28KB	0.58	12.52
	Classifier	19.3	19.4	36KB	0.58	18.23
	Return	18.0	18.1	16KB	0.59	6.77
Images	Distribute	18.0	18.1	632KB	0.59	0.42
	Recognize	26.6	27.1	83.2MB	0.63	24.79
	Combine	18.0	18.1	2.5MB	0.59	0.36
	Translation	25.3	386.9	29.1MB	2.34	26.65

Table 4.3: Breakdown of memory size and compute statistics per module per workload. Load Size: the size of the loaded module before execution, Inited Size: module size after initialization. Init Time: module initialization time. CPU Time: Processing time of enclave (seconds), CPR size: data copied/zeroed on checkpoint restore. "Images: Recognize" reports the maximum of all four image recognition enclaves.

Inputs. We designed workload inputs to be realistic. Email bodies are from a spam training set [spa]. Email attachments are a set of PDFs randomly attached to 30% of emails (and that figure is from a study of corporate email characteristics [ema]). Images are a mix of photographs, computer-generated patterns, and logos. Gene data was synthesized based on DisGeNet [dis]. Translation text comes from the News Commentary dataset [wmt].

Confinement overhead. In Figure 4.7, the Sandbox and Sandbox+Enc overheads are necessary for confinement, and across all workloads, encryption does not add significant overheads. For Genes, the confinement overhead is

		Event Counts (Thousands)		
		System Calls	Page Faults	Interrupts
Email	Distribute	47	60	0.47
	DSPAM	1,290	1,810	6.00
	ClamAV	247	423	7.00
	Combine	120	2	0.08
Health	LoadModel	82	280	56.00
	Classifier	1,840	359	151.00
	Return	668	162	3.00
Images	Distribute	2	2	0.04
	Recognize	88	174	6.00
	Combine	14	3	0.13
	Translation	303	248	8

Table 4.4: Enclave exits (System Calls, Page Faults, and Interrupts) per workload per module. "Images: Recognize" reports the maximum of all four image recognition enclaves.

high (100%) because it runs a very simple SVM classifier. The actual data processing time is small, which amplifies the effect of Ryoan's data buffering/-padding and serves as a worst-case scenario. For Images, the workload involves heavy computation with OpenCV, and the confinement overhead is 18%.

Checkpoint restore overhead. The CPR Size column in Table 4.3 shows the amount of memory copied/zeroed on checkpoint restore. Figure 4.7 (the difference between the Sandbox+Enc+Marsh and Sandbox+Enc+Marsh+CPR columns) shows that checkpoint restore's impact on performance is significant (55%) for Genes because it has the lightest per-unit workload (≈1ms) and the relative cost of page fault handling is high; in contrast, its impact on Images is only 3%, which has the heaviest per-unit workload (≈2s).

SGX overhead. Executing code in an SGX-protected enclave imposes several overheads. We simulate SGX hardware overheads by using delays to model the performance of SGX instructions, and flush the TLB on all enclave exits (we could not directly measure execution on our hardware because it lacks SGX version 2 features (§4.4.1)). Besides explicit `EEXIT` instructions, we also model enclave exits due to events like exceptions and interrupts (Table 4.4). We measure a hardware delay of 3.9μs for each `EENTER` /`EEXIT` pair, and 3.14μs for each `ERESUME` /Async-Exit pair.

We also measured SGX instruction costs on the more recent and powerful Intel Xeon E3-1270 v6 3.80 GHz processor. On the Xeon processor, `EENTER` /`EEXIT` pairs cost 2.34μs, and `ERESUME` /Async-Exit pairs cost 1.85μs. This processor's clock rate is about 65% faster than the laptop, and the SGX costs have been reduced by about that factor.

Version 2 instructions `EACCEPT` , `EMODPE` , `EMODPR` are simpler than `EENTER` and `EEXIT` , so we model their cost at one-tenth of one `EENTER` /`EEXIT` pair. Figure 4.8 explores the effect of varying this cost on the runtime of our workloads. If the version 2 instructions turn out to be as costly as an `EENTER` /`EEXIT` pair (3.9μs), for instance, the running times of our email, health, images, and translation workloads increase by 25%, 14%, 7%, and 4% respectively.

Every checkpoint-related page fault requires one `EMODPE` to extend page permissions. Every page reverted after checkpoint requires one `EMODPE` followed by one `EACCEPT` . Unfortunately, version 2 of SGX also imposes ad-

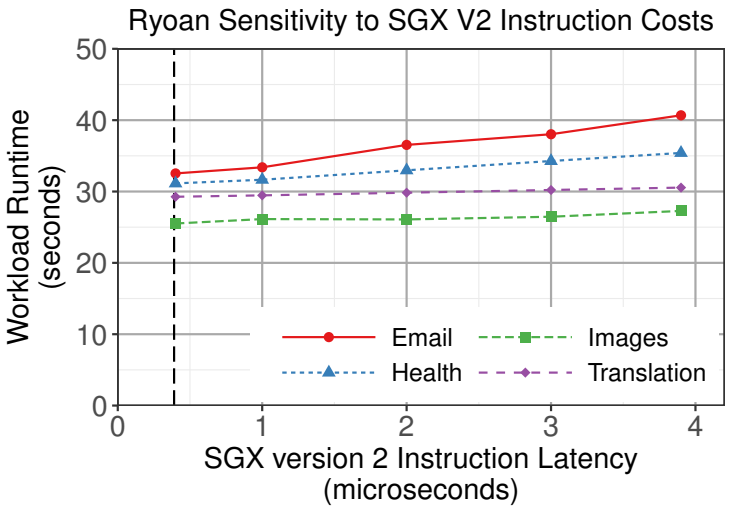

Figure 4.8: Ryoan application workloads' sensitivity to emulated instruction cost. The dashed vertical line denotes the delay (0.39μs) used to compute the Ryoan bars in Figure 4.7.

ditional synchronization (via extended behaviors of `ETRACK`) when modifying the enclave's page state [MAA+16]. We believe the performance effect on these workloads will be negligible, given that our applications only have one thread per enclave. SGX execution also requires syscall marshaling to copy system call arguments and results to and from untrusted memory, but we measure the marshaling overhead as negligible. All results are shown in Figure 4.7.

Checkpoint restore vs. initialization. Creating an enclave and loading a module takes less than 0.5s for all our cases. However, Table 4.3 shows application-level initialization times are over 20 seconds for DSPAM and ClamAV because they need to load and parse databases. As a result, for this workload, it is preferable to use Ryoan's checkpoint-based reset rather than

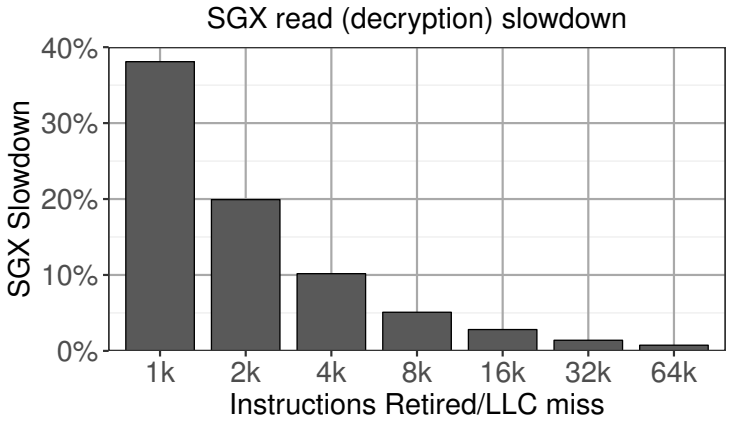

Figure 4.9: Slowdown observed with respect to LLC read-misses running the cache-miss microbenchmark inside an SGX enclave versus running the same code without SGX.

re-initialize the modules for every work unit. Enclave construction imposes further overheads on re-initialization. Even the creation of small enclaves (e.g., 298KB) incur a penalty of 30 milliseconds. In comparison, Ryoan's checkpoint-based reset is much more efficient, and the per-unit cost is under 10ms.

4.6.2 SGX encryption overheads

Enclave memory is encrypted whenever it leaves the processor. This invariant means additional operations are required when the processor reads memory from RAM: encrypt on write, decrypt on read. These additional operations add latency to last level cache (LLC) misses. Encryption related performance penalties are absent from our performance model; here, we explore their cost.

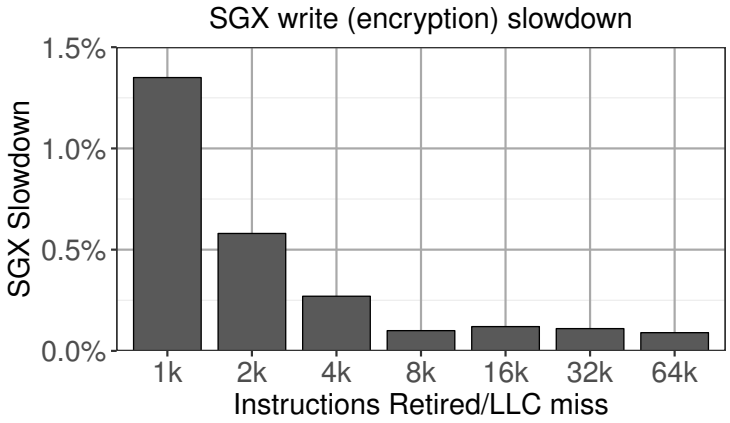

Figure 4.10: Slowdown observed with respect to LLC write-misses running the cache-miss microbenchmark inside an SGX enclave versus running the same code without SGX.

LLC miss microbenchmark. To measure the memory controller overheads of SGX, we use a microbenchmark that executes a fixed number of instructions per cache-miss (read or write). We execute the same microbenchmark as part of a normal process and compare it to execution in an enclave protected by SGX. The slowdown incurred by running the microbenchmark in an SGX enclave for a varying number of retired instructions (and for read or write LLC misses) is shown in Figures 4.9 and 4.10. When computation does not access memory (and we have a large number of instructions per cache-miss), the enclave code's performance is very similar to unshielded execution. The microbenchmark makes no system calls, and we eliminated page faults by ensuring all enclave memory is touched before measurement begins. Therefore the only enclave exits are due to interrupts, and their effect on the total time is insignificant.

Benchmark	Module	Instructions/LLC miss Read Miss	Write Miss	Memory controller SGX slowdown
Email	ClamAV	1,260	5,090	32.0%
Health	Classifier	14,310	24,650	3.2%
Images	Recognize	32,760	9,000	1.4%
Translation	(one module)	12,560	34,510	3.5%

Table 4.5: Instructions per LLC miss on Ryoan benchmarks. Memory controller SGX slowdown is the slowdown measured for microbenchmarks of equivalent miss patterns on SGX hardware.

Write-misses are cheap because the processor does not wait for the memory controller to encrypt data. A last-level write cache-miss every 1,000 instructions incurs about 1.4% execution time overhead. Read-misses can cause significant delays, and programs with high read-miss rates will run slowly within an enclave. A read-miss every 1,000 instructions causes a 38.1% performance overhead, which falls to 10.1% once the read-misses happen every 4,000 instructions. The processor often needs the data from a read before it can do any useful work and, therefore, will stall waiting for the data to be decrypted.

To understand the slowdown that would be incurred for Ryoan's benchmarks due to SGX encryption overheads, we measure the number of instructions retired per LLC miss, reported in Table 4.5. We focus on the enclaves which dominate the performance of the benchmarks.

The "Memory controller SGX slowdown" column reports our projected enclave slowdown based on the workloads LLC rate and measurements of our LLC miss microbenchmark. Email shows the largest effect at a projected 32% slowdown, much higher than the other benchmarks. The other applications

execute large numbers of instructions for every last level miss, putting our estimate of SGX encryption overheads at less than 5%.

Chapter 5

Telekine: Secure Computing with Cloud GPUs

GPUs have become popular computational accelerators in public clouds. Accuracy improvements enabled by GPU-accelerated computation are driving the success of machine learning and computer vision in application domains such as medicine [Hem17,SFB+15] transportation [NVIb], finance [GGKSC13], insurance [NVI16], video games [SSR], and communication [NVI17a].

Trusted execution environments (TEEs) should, in principle, make the cloud an option for users who refuse to trust the provider. Researchers have proposed GPU-based TEEs [VVB18] and TEE extensions for GPUs [JTK+19], though none have been built or deployed. However, as we argue below, a design that simply composes components that run in hardware-supported CPU and GPU TEEs will fail to provide strong security due to side channels.

GPU-accelerated applications have three main software components:

This chapter is based on a previous publication: "Telekine: Secure Computing with Cloud GPUS", by Tyler Hunt, Zhipeng Jia, Vance Miller, Ariel Sezekely, Yige Hu, Christopher J. Rossbach, and Emmett Witchel in the proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI), February, 2020, Santa Clara, California, USA [HJM+20]. My contributions to this publication include designing and implementing Telekine's data-oblivious streams, and developing the arguments around GPU trusted execution environments.

(1) an API and a user library (e.g., CUDA [NVIa] or HIP [HIP]) that provides high-level programming functionality and executes on a CPU; (2) CPU-side control code at the user and the system level that manages communication with the GPU, and (3) GPU kernels (programs) that execute on the GPU device itself. It is the data and code that moves between the CPU and GPU that potentially creates side channels visible to CPU-side code.

An attacker can extract meaningful information from the execution time of code on the GPU; through control of privileged software, a cloud provider can easily compute these execution times on the CPU by observing communication with the GPU. For example, we demonstrate a novel attack on image-recognition, machine-learning models that allows malicious system software to correctly classify images from ImageNet [DDS+09] used as input to the model. By observing only the timing of a model trained to classify images (the image model), we build a new model (the timing model) that classifies images based on the execution timing of layers in the image model. Even if a security-conscious user encrypts their input images (and decrypts them on the GPU), a system administrator can use the GPU kernels' timing information (measured on the CPU) from the image model to classify the input images anyway. We train the timing model to distinguish images of two classes with 78% accuracy. For more classes, accuracy decreases but stays above random guessing.

We propose Telekine, a system that enables the secure use of cloud GPUs without trusting the platform provider. GPU TEEs provide a secure

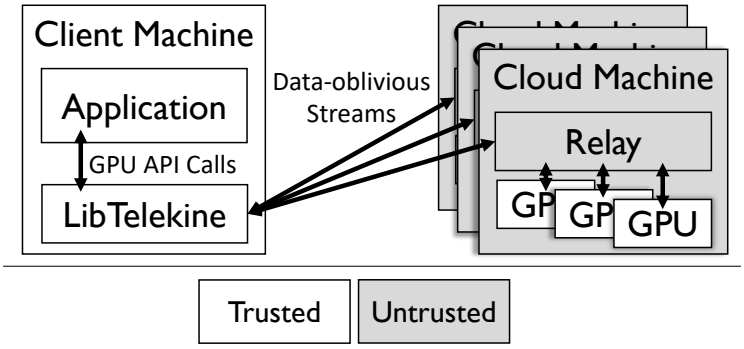

Figure 5.1: Telekine components and their organization.

execution environment but leave the user open to side channels when communication depends on secret data. Telekine makes communication with the GPU TEE data oblivious, completely independent of secrets in the input data. Data obliviousness is a strong property that excludes the existence of side-channel attacks against CPU-side code and host/device communication whose observable behavior (e.g., timing, memory accesses, DMA sizes, etc.) depends on secret input data.

Telekine has three components (shown in Figure 5.1): libTelekine that runs on a trusted user machine (a *client*), GPUs physically attached to a cloud machine (a *server*) that supports GPU TEEs with specific security requirements (§3.3.2), and the relay which facilitates communication between libTelekine and the GPU. Telekine uses a GPU TEE because it needs a mechanism to protect GPU computation from the cloud provider; a GPU TEE is tailored to that task.

Telekine protects the application and GPU runtime by moving it from

the cloud to the client. The advantage of this approach is that the user must already trust their client machine; the application and user libraries are large and complex and, therefore, prone to side-channel attacks, making them difficult to secure if they execute in the cloud. The disadvantage is that GPU libraries assume a local GPU with a fast, high-bandwidth connection to the CPU. Telekine decouples the user library from low-level GPU control by interposing on the GPU API and efficiently forwarding API calls to the server (a technique known as API remoting). API remoting has often been used to virtualize GPUs [YPAR20, GMAC10, GGS+09, SCS09, VSB14, BBNLS10, DIM+09, DPS+11, KSL+12, LNEAEG11, LC11, Bit], but to our knowledge has never been used for security. A client using Telekine does not need to have a GPU installed.

Telekine treats the CPU-side control code on the cloud server ("Relay" in Figure 5.1) as completely untrusted, almost as if it were part of the network. The client machine establishes a cryptographically secure channel directly with the code executing on the cloud GPU. The network and the CPU-based code on the server can delay the computation, but cannot compromise its privacy or integrity.

Telekine secures the communication between the client machine and the cloud GPU by transforming the user's GPU API calls into *data-oblivious streams*. Data-oblivious streams are similar to constant time defenses [ANB+18] in that they aim to remove timing channels by ensuring that observable events are deterministic regardless of secrets. Telekine constructs data-oblivious streams

by reducing all API calls to a sequence of code execution (`launchKernel`) and data movement (`memcpy`) commands. It then schedules these commands at a fixed rate, possibly creating new commands, or splitting `memcpy` commands into fixed-size pieces. Fixed-sized, fixed-rate communication is data oblivious; it ensures that any observable patterns are independent of the input data and, therefore, devoid of side-channel information. Fixed-rate communication is not a novel way to eliminate side channels, but Telekine's design shows how to apply it efficiently to modern GPU-based computing.

Given that Telekine requires a GPU TEE, it is logical to wonder why it does not use a CPU TEE. After all, putting the application and programming libraries into a CPU TEE would reduce the latency and increase the bandwidth for communication between libTelekine and the GPU. Unfortunately, Intel and ARM TEEs do not prevent side channels as part of their threat model [Joh17, PS19]. Keystone [LKC+18] and Komodo [FBHP17] intend to address side channels for RISC-V and ARM, respectively, but work is ongoing. Also, making existing applications data oblivious is difficult for programmers, requires access to source code (not needed by Telekine), and often slows down a program greatly (e.g., Opaque [ZDB+17] slows down data analytics by 1.6–46×). Should future CPU TEEs evolve to address side channels, Telekine can use them. Much of Telekine focuses on securing the communication between trusted components, which can be an improved CPU TEE and a GPU TEE, or they can be the client machine and server GPU TEE, as they are in our prototype.

Contributions. Telekine is the first system to offer efficient, secure execution of GPU-accelerated applications on cloud machines under a strong and realistic threat model. We use Telekine to secure several GPU-accelerated applications via two frameworks: the MXNet [CLL+15] machine learning framework and the Galois graph processing system [PP16]. On a realistic testbed, Telekine provides strong secrecy and integrity guarantees, including side-channel protection. MXNet [CLL+15] training for three different, modern image recognition models incurs a 10–22% performance penalty relative to a baseline with a locally attached GPU. MXNet inference for the same models over a connection from Austin, TX to the Vultur's Dallas, TX datacenter [Vul] incurs a penalty of 0-8% for batch sizes of 64 images. Telekine runs graph algorithms using Galois [PP16] on one and two GPUs with 18%–41% overhead.

This paper makes the following contributions.

- We demonstrate a CPU-side timing attack on deep neural networks that allows a compromised OS to correctly classify images in encrypted input (§5.3).
- We provide a design and prototype for Telekine, a system that eliminates CPU-based side-channel attacks against a GPU TEE with a novel variant of API remoting to execute secret-dependent code on the GPU TEE and a trusted client (§5.4).
- We thoroughly evaluate the performance, robustness, and security of Telekine, protecting a variety of important workloads on one and two GPUs: machine learning and graph processing (§5.6).

5.1 Telekine speciation of the malicious public cloud threat model

In all current cloud GPU platforms, the cloud provider's privileged software, and hence administrators, can gain easy access to GPU state, creating a significant attack surface including explicit channels such as GPU memory, firmware, and execution context. Work in this area agrees on the vulnerability of any program state on the GPU to privileged software [VVB18, JTK+19].

Telekine assumes a GPU TEE, with capabilities similar to current research proposals like Graviton [VVB18]. The details can vary, but a GPU TEE establishes secure memory on the GPU device and provides a protocol to initiate a computation that can be remotely attested to start from the correct state (code and initial data) and execute privately and without interference from the CPU side. We provide additional detail on Telekine's TEE requirements in Section 3.3.2.

GPU TEEs do not (by themselves) secure communication with the CPU, and our attack (§5.3) shows how much information there is in the precise timing of CPU/GPU communication. Telekine protects communication with the GPU, guaranteeing that the adversary cannot learn about input data directly or through side channels, including timing channels.

While secure control of a GPU has been proposed [VVB18, JTK+19], there has been little work securing side channels. These side channels undercut the security of the TEE. In addition to the timing attack we developed (§5.3), AES key extraction using shared GPU hardware [JFK17,JFK16,GESM17] has

been demonstrated. And recent side-channel attacks [NNQAG18] have shown practical methods to fingerprint websites using performance counters observed during GPU rendering in the browser.

5.1.1 Guarantees

Telekine provides the following secrecy properties, which prevent explicit or implicit data flow from input data to an external observer.

S1 (content): Messages are encrypted to ensure an observer cannot directly read their content.

S2 (timing): The transmit schedule for messages is fixed. Any transmission delays are independent of input data.

S3 (size): The size of each message is fixed. Telekine pads and/or splits messages to achieve fixed-sized messages.

Telekine also provides the following integrity properties to ensure that any result the user receives is either a result that could have been generated by a GPU hosted by a completely benign cloud provider, or an error.

I1 (content): The content of all communication is protected by an end-to-end integrity check; a message authentication code (MAC) allows Telekine to detect modifications, returning an error if any are detected.

I2 (order): Each message carries a sequence number which allows Telekine to detect out of order messages. The sequence numbers also prevent replay attacks.

I3 (API-preserving): Commands issued by the application should affect

GPU state in the same way they would on a local GPU, regardless of any transformations that Telekine applies.

GPU commands have semantics that Telekine must maintain for correctness. For example, GPU runtimes expose a *stream* [NVI17b] abstraction to application code. API calls issued by the application on the same stream are executed serially in the order they were issued. A kernel launched from a particular stream will block the completion of subsequent API calls on that stream until that kernel terminates. Applications can have many streams which map to different command queues exposed by hardware. API calls made on separate streams can be executed in parallel. Telekine must respect the data dependence semantics of streams.

5.1.2 Limitations.

Physical side channels and denial of service attacks are out of scope. In situations where an adversary monitoring physical side channels like temperature [MRR+15], power [KJJ99], or acoustical emanations [CLL+17] is a concern, Telekine would need to be augmented with other techniques to maintain security. In our threat model, a cloud provider wishing to deny service can always do so, e.g., by interrupting the network or refusing to run user processes.

Telekine provides clients a mechanism to disguise their end-to-end runtime but does not impose policy. Applications can choose the most efficient policy for their security needs. We believe end-to-end runtime is a poor pre-

dictor of input data (and our experiments in Section 5.3 bear this out), further justifying the clients setting policy.

5.2 GPU Trusted Execution Environment requirements

Telekine assumes GPU TEE support similar to Graviton [VVB18] to prevent MMIO access to GPU status and configuration registers during secure execution. Due to Telekine's focus on side channels, it has requirements beyond the previously proposed GPU TEEs. These requirements are more straightforward to provide than the core TEE functionality.

Eliminate GPU side channels. Some TEE designs allow different tenants/principals to execute concurrently (e.g., SGX, Keystone), sharing the underlying hardware. Concurrent execution is attractive from a utilization perspective, but it provides a rich side-channel attack surface that has plagued the security of CPU TEE designs. Telekine assumes side channels from concurrent principals (e.g., memory access timing and bandwidth) do not exist on the GPU TEE. A conservative design that prevents hardware side channels is to disallow concurrent execution. Graviton TEEs scrub their state (e.g., registers, memory, caches) after resources are freed, so there is no danger of tenants observing transient state from any previous computation.

Conceal kernel completions. GPUs signal the CPU via an interrupt when a kernel has completed its execution. Interrupt timing leaks information about

the kernel's runtime. Rather than rely on interrupts, Telekine uses data-oblivious streams (§5.4.1) that include tagged buffers that allow the GPU to communicate computational results back to the client. The platform only sees DMA from the GPU to untrusted CPU memory at a fixed rate.

Support no-op kernel launches. Dependences between GPU kernels often cause the launch of one kernel to wait for another's completion, which provides indirect timing information. The GPU TEE must support a no-op kernel launch command so that Telekine can generate cover traffic to ensure the adversary sees kernel launches at a fixed rate.

Timely command consumption. The GPU TEE should consume its command queue independent of how long kernels execute on the GPU. If the GPU waits until each kernel completes before dequeuing the next launch command, it can fall behind the input queue fill rate, allowing the input queue to fill. The adversary can detect this situation by observing how often the encrypted queue content changes, creating a proxy for kernel execution time. The GPU should consume command queue entries at a fixed rate, discard the no-ops, and store the real commands internally until the can execute them. Telekine can hold back real kernel launches and send no-op launches in their places to ensure these internal GPU queues do not fill up.

5.3 Example side-channel attack

Telekine addresses software attacks launched by an adversary resident on a cloud host, such as those launched by a malicious system administrator or a network-based attacker who has compromised the platform's privileged software. These attacks use privileged software to compromise the privacy or integrity of user code and data. Telekine is particularly focused on protecting against timing channels because effective, general-purpose attacks using timing channels have recently been demonstrated at the architecture level [KGG+18, LSG+18,VBMW+18,SLM+19], the OS level [vSGBR18,XCP15], and the GPU programming level [JFK17,JFK16]. Modern CPU TEEs exclude side channels from their threat model [Joh17, PS19, GESM17], leaving current hardware-supported security primitives vulnerable to side-channel attack. Telekine offers a unique and efficient security solution for cloud resident, GPU-based computation.

We demonstrate a proof-of-concept attack on machine learning inference in which the adversary uses the execution timing of individual GPU kernels to learn information about encrypted input data. Our attack allows privileged software on the cloud host to correctly classify images using only the timing of GPU kernel execution obtained on the CPU. The attacker can train their timing model on their own input; they do not need the victim's training data. The image data remains encrypted while on the CPU, and the attack does not require any access to GPU architectural or microarchitectural state (including GPU timers).

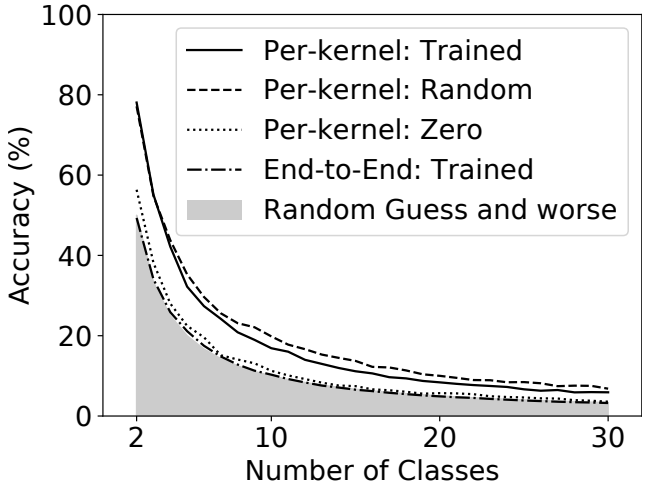

Figure 5.2: Accuracy of multiclass classification for side-channel attacks for increasing numbers of input classes.

Attack basics. Convolutional neural networks (CNNs) are a popular neural network architecture for analyzing images [HZRS16, SVI+16, HLWvdM17]. Each network consists of multiple layers, including convolutions, which are good at detecting the input image features that the remainder of the network can use to classify the image. When CNNs are executed on a GPU, the computation for each layer roughly corresponds to the execution of a single GPU kernel. While the actual mapping between layers and kernels is often more complex, the intuition behind our attack is that the timing of the execution of certain CNN layers (and hence their GPU kernels) indicates the presence or absence of certain features within the input image. This mapping makes the per-layer execution time itself a rich feature.

Telekine defeats the attack by removing the adversary's ability to infer the timing of individual kernels. The adversary retains only the ability to

measure the end-to-end runtime of the inference task. However, our data show that end-to-end runtime provides very little predictive value, making the attack not much more accurate than randomly guessing (Figure 5.2). Telekine gives users the mechanism to disguise their end-to-end execution time, should they decide to do so (§5.1.2).

Attack details. We demonstrate this attack on ResNet50 [HZRS16], a CNN widely used for image recognition, using the timing of GPU kernel completion events as detected by the operating system on the CPU (though we monitor a function in the GPU's user-level runtime for ease of implementation). We evaluate the accuracy of our attack using 5-fold cross-validation.

We start with a pre-trained model for the standard ImageNet [DDS+09] dataset, which contains 1,000 different image classes. Figure 5.2 shows the accuracy of distinguishing image classes based on the timing of the pre-trained model's layers (Per-kernel: Trained), versus the same attack using only end-to-end timing information (End-to-end: Trained). The accuracy of the per-kernel classifier is startlingly good for small numbers of classes: 78% for two classes, 55% for three, and 42% for four. As the number of classes of input images increases, the accuracy of our classification declines, but it remains much better than random guessing, outperforming guessing by over 1.9× even among 30 input image classes.

We believe the root cause of the attack is timing dependent GPU operations, probably multiply by zero. We compare a pre-trained model (Per-kernel:

Batch purity	Accuracy
0.25	29.3%
0.4	33.0%
0.6	41.0%
0.8	50.0%
0.9	56.1%
1.0	65.4%

Table 5.1: Accuracy distinguishing four classes with batches of size 32, varying the percentage of each batch containing images from the target class.

Trained with no zero-valued weights), a randomly initialized model (Per-kernel: Random with 0.2% zero-valued weights), and a model whose weights are all zero (Per-kernel: Zero with 100% zero-valued weights). The zero model has bad accuracy that is close to random guessing. A randomly initialized model is best, followed by the pre-trained model.

We generated these results using MXNet [CLL+15] ported to HIP on the ROCm version 1.8 stack for AMD GPUs, the version used in the prototype; we saw similar results on the 2.9 version. Preliminary tests showed that this specific attack is much less powerful on NVIDIA GPUs.

Batched classification. Because inference is often done in batches, we examine the accuracy of a batched attack. We construct batches by splitting each ImageNet class into disjoint training and test sets. Images are then randomly sampled from each of these sets to form the batches.

We present the accuracy of our attack when distinguishing four ImageNet classes in batches of size 32 (Table 5.1.) Each batch consists of the

given fraction of images from a primary-class (Purity), and randomly selected images from the remaining three classes. Our objective is to correctly identify the primary class.

Batches help, with the accuracy of our attack improving with larger batch sizes. Larger batches execute more operations, effectively amplifying the timing signal our attack relies on. Moreover, larger batches smooth out execution timings for outlier images, which would otherwise be less recognizable to our attack model. When distinguishing four classes (Table 5.1), the batched attack is better than random guessing even when only 25% of the input images come from the target class. The accuracy increases with higher batch purity, outperforming single images by up to 64%.

5.4 Design

Telekine secures GPU-based computation from active attackers, including side-channel threats. Side channels include the execution timing of individual GPU kernels and data movement to and from the GPU. Telekine achieves its security by transforming an application's computation so that all communication—including data movement—among trusted components is data oblivious. Telekine only trusts the client machine and the in-cloud GPU TEE. Therefore, it must efficiently coordinate the computation between these entities, even though communication occurs over a wide area network, rather than over higher-bandwidth, lower-latency fabric like a data center network or a PCIe bus.

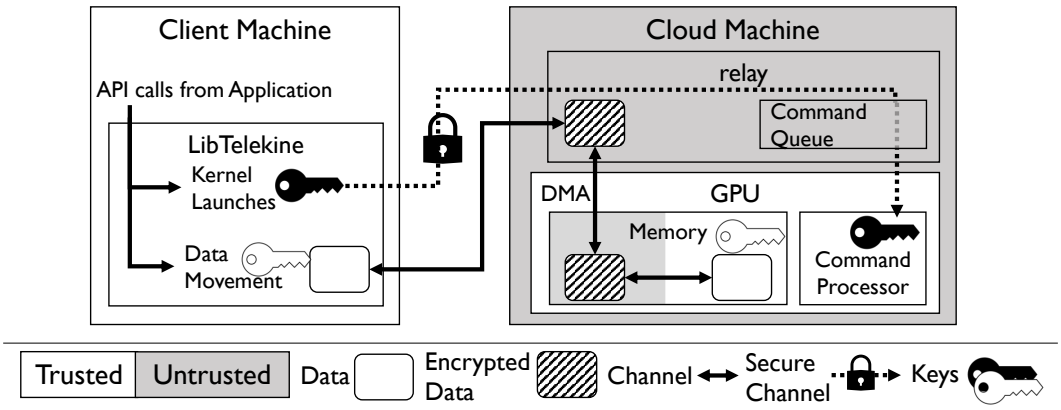

Figure 5.3: Detailed Telekine overview.

Telekine consists of three components (depicted in Figure 5.1, with detail in Figure 5.3).

- LibTelekine: a library that intercepts GPU API calls from the application and transparently transforms them into a data-oblivious command stream.
- Relay: an untrusted process that runs in the cloud and directs the client's command stream to the GPU.
- GPU: a GPU (or multiple GPUs) with TEE support that meets Telekine's requirements (see §3.3.2 for details).

LibTelekine is linked into the application running on the client. During its execution, the application issues a stream of GPU commands through the normal GPU API. Similar to normal API remoting [DPS+10, VSB14, Bit], libTelekine redirects API calls made by the client to a server process with a GPU runtime–the relay on the cloud machine. Telekine treats the relay almost

as if it were part of the network, relying on it to communicate with the GPU but protecting that communication with end-to-end techniques. The relay is not part of Telekine's trusted computing base.

Authenticated encryption (AES-GCM [Dwo07] in our prototype) and sequence numbers protect all communication between libTelekine and the GPU. This protection creates a secure channel satisfying the secrecy property *S1 (content)* and the integrity properties *I1 (content)* and *I2 (order)* (described in §5.1.1), ensuring that the GPU commands issued by libTelekine can only be read by the GPU, and any tampering or reordering is detectable. However, by observing when messages are exchanged with the GPU (regardless of whether they are encrypted), the adversary can get timing information about the computation on the GPU.

Telekine's goal is to remove all timing information from the encrypted stream of GPU commands. It removes timing information by sending commands (GPU runtime API calls like `launchKernel` and `memcpy`) at a fixed rate, independent of input data. Fixed-rating is a simple idea, but Telekine must overcome two major challenges to fix-rate GPU communication.

1. Different GPU command types are distinguishable because they have different sizes, and they result in different communication patterns with the GPU. (e.g., `launchKernel` commands interact with MMIO ring buffers and `memcpy` commands are handled using DMA). Telekine must ensure that the attacker's ability to distinguish between these commands conveys no information about the input data.

2. Conventional GPU command streams (§5.1.1) exhibit a variety of data-dependent behavior whose timing is externally visible (e.g., a kernel launch after a data transfer will wait for the data transfer to finish). Telekine must maintain the ordering semantics induced by such data dependencies.

Telekine introduces a new primitive to overcome these challenges: *data-oblivious streams*. Data-oblivious streams transparently replace conventional GPU streams (and applications may have more than one), maintaining their semantics while making their communication with the GPU data oblivious. First, they separate commands by type and schedule each type independently. Second, they split, pad, and batch commands of each type so that the encrypted payload is always the same size for messages of that type, satisfying *S3 (size)*. Third, they inject management commands as needed to maintain data-dependencies across message types, satisfying *I3 (API-preserving)*. Finally, data-oblivious streams send the transformed commands according to a fixed schedule, satisfying *S2 (timing)*.

The relay, privileged software on the cloud machine and the network stack can delay commands since they are under complete control of the (possibly adversarial) cloud provider. However, they cannot delay commands in a way that leaks input data because all observable behavior of the trusted computing base (including its timing) is independent of input data.

5.4.1 Data-oblivious stream construction

Constructing data-oblivious streams only requires reasoning about `memcpy` and `launchKernel` commands. The TEE takes care of initialization (§3.3.2). The only other runtime commands deal with stream synchronization, and Telekine transforms those commands into `memcpy` and `launchKernel` commands as well (discussed fully in §5.4.4). `memcpy` commands are visible to the untrusted host's privileged software because GPU drivers use DMA for efficient data transfers. In Telekine, the data itself is protected and copied to/from a fixed staging area in untrusted GPU memory, so the destination/source of the `memcpy` does not leak information.

Conventional GPU streams can create timing channels from `memcpy` and `launchKernel` commands because a `memcpy` command waits for all previous `launchKernel` commands on the same stream. To eliminate this channel, Telekine uses two GPU streams to construct a single data-oblivious stream. Telekine uses one GPU stream to launch the application's kernels; this stream is called the ExecStream. Telekine uses the other stream—called the XferStream—to move data to and from the GPU. Telekine ensures that commands on the XferStream never leak information about the kernel execution time by waiting for commands on the ExecStream.

The ExecStream. Application kernels are all launched on the ExecStream. LibTelekine maintains a queue of the `launchKernel` commands requested by the application and releases the commands in order according to the fixed-rate

schedule. The GPU consumes these commands independently of any ongoing kernel execution and buffers them internally since their execution must be serialized according to GPU stream semantics. Telekine honors data dependencies between `memcpy` and `launchKernel` commands by inserting data management kernels that block the progress of the ExecStream by spinning until the data is in place.

The XferStream. Telekine launches data transfers requested by the application on the XferStream. Unlike `launchKernel` commands, `memcpy` commands are directional (i.e., client-to-GPU and GPU-to-client), and directions are detectable. For example, because the adversary can observe interaction with the network, it can differentiate between messages that came over the network in transit to the GPU, and messages copied from the GPU to be sent over the network. LibTelekine maintains separate queues for each direction and schedules them independently to avoid leaking information. Data for client-to-GPU transfers starts on the client, flows through the relay, and into untrusted memory on the GPU. LibTelekine then enqueues a kernel, which moves the data from the untrusted staging memory into trusted GPU memory. Similarly, in the GPU-to-client direction, Telekine first enqueues a `launchKernel` on the XferStream to move the data into untrusted GPU memory, then issues a `memcpy` to copy it to the relay where it can be transferred over the network back to the client.

Fixed-size commands. Telekine ensures that all `memcpy` commands are the same size by splitting and padding the `memcpy` commands issued by the application to a standard size. When there are no pending `memcpy` commands, Telekine maintains the same data flow rate by scheduling dummy, standard-sized `memcpy`s to/from a staging buffer. Similarly, Telekine pads all `launchKernel` commands are to the same size (320 bytes in our prototype). When no `launchKernel` command is available, Telekine schedules no-op `launchKernel` commands.

Schedules. Any schedule Telekine uses for GPU communication is secure so long as it does not depend on the data being protected. Our prototype uses simple schedules which send a fixed number of fixed-sized commands after each fixed-time interval. For instance, Telekine might launch 16 kernels on the ExecStream every three milliseconds, and send then receive 4MB of data every six milliseconds on the XferStream.

Schedules can leak the category. While scheduling work at a fixed rate is a well-known technique to avoid side-channel leakage, the exact schedule is relevant to performance. We report our schedules in Table 5.2, and they are the same for all tasks of a given category, e.g., training different machine learning models with MXNet. However, they can differ across categories, e.g., Galois has a different ExecStream schedule from MXNet (§5.6). Under our threat model, the adversary would be able to differentiate these workloads

Algorithm 1 Telekine's replacement functions for `memcpy` and `launchKernel`. Splitting and padding steps are omitted for brevity.

```
 1: function LaunchKernel(kern, args...)
 2:     Enqueue(kernelQueue, {kern, args})
 3: end function
 4:
 5: function MemcpyH2D(src, dst)
 6:     buf ←ChooseTaggedBuffer()
 7:     LaunchKernel(copy_in, buf, dst)
 8:     Enqueue(dataQueueH2D, {src, buf})
 9: end function
10:
11: function MemcpyD2H(src, dst)
12:     buf ←ChooseTaggedBuffer()
13:     LaunchKernel(copy_out, src, buf)
14:     Enqueue(dataQueueD2H, {buf, dst})
15: end function
```

from their network traffic. A user can always choose a more generic, but lower performing schedule if this is a concern.

5.4.2 Telekine operation

Algorithm 1 and Algorithm 2 provide a high-level description of Telekine's data-oblivious streams. In Algorithm 1, Telekine intercepts the application's calls to `launchKernel` and `memcpy` and transforms them into interactions with queues: kernelQueue, dataQueueH2D, and dataQueueD2H (splitting, padding, and encryption steps are omitted for brevity). The Telekine threads shown in Algorithm 2 dequeue the commands and release them to the GPU according to the schedule. Telekine waits at lines 7, 18, and 29 for the next available time slot, ensuring that interactions with the queues do not influence the messages'

Algorithm 2 Periodic tasks performed by Telekine according to the schedule. Encryption and decryption steps are omitted for brevity.

```
1: loop                                                    ▷ ExecStream Thread
2:     if EMPTY(kernelQueue) then
3:         op ←no_op
4:     else
5:         op ←DEQUEUE(kernelQueue)
6:     end if
7:     WAITFORSCHEDULEDTIME()
8:     REMOTELAUNCHKERNEL(op)
9: end loop
10:
11: loop                                  ▷ XferStream Client-to-GPU (H2D) Thread
12:     if EMPTY(DataQueueH2D) then
13:         src ←dummy_CPU
14:         dst ←CHOOSETAGGEDBUFFER()
15:     else
16:         {src, dst} ←DEQUEUE(dataQueueH2D)
17:     end if
18:     WAITFORSCHEDULEDTIME()
19:     REMOTEMEMCPY(src, dst)
20: end loop
21:
22: loop                                  ▷ XferStream GPU-to-Client (D2H) Thread
23:     if EMPTY(DataQueueD2H) then
24:         src ←CHOOSETAGGEDBUFFER()
25:         dst ←dummy_CPU
26:     else
27:         {src, dst}←PEEK(dataQueueD2H)
28:     end if
29:     WAITFORSCHEDULEDTIME()
30:     REMOTEMEMCPY(src, dst)
31:     if dst ≠ dummy_CPU then
32:         if TAGMATCHES(dst) then
33:             DEQUEUE(dataQueueD2H)
34:         end if
35:     end if
36: end loop
```

timing.

Most `memcpy` commands have strict ordering requirements with respect to kernels that operate on their data. The `memcpy` then `launchKernel` idiom ensures that the launched kernel has fresh data to process. While Telekine decouples `memcpy` commands by scheduling them on their own stream for security, it needs to preserve the original ordering semantics expected by the application. Telekine maintains these semantics by injecting its own data management kernels into the ExecStream (shown on lines 7 and 13 of Algorithm 1) to enforce the ordering expected by the application. These data management kernels operate on *tagged buffers*, which Telekine uses to synchronize data access.

Tagged buffers. Tagged buffers are pre-allocated staging buffers on the GPU, each with an associated tag slot. Telekine assigns every `memcpy` operation a tagged buffer and a unique tag, represented by "ChooseTaggedBuffer" in Algorithm 1 and Algorithm 2. Data management kernels producing data (e.g., copying out the result of a kernel computation) write the tag into the tag slot of the chosen tagged buffer after the operation has completed and a memory barrier completes. Data management kernels that consume data (e.g., some kernels wait for data a kernel expects to use as input) wait until the tag slot of the assigned buffer contains the expected value. They cannot be sure the buffer data is valid until the tag value matches its expectation.

Data management kernels. Telekine inserts its own data management kernels into the ExecStream; these kernels either produce or consume tagged buffers depending on the direction of the transfer. There are two kernels: `copy_in` and `copy_out`. Both kernels take an application-defined memory location, a tagged buffer, and a tag as arguments. For CPU-to-GPU `memcpy`s, libTelekine inserts a `copy_in` launch into the ExecStream. The `copy_in` will repeatedly check the tag slot of the buffer; completing the copy to the application's buffer only after verifying the tag slot matches the tag it was given as an argument. To service GPU-to-CPU `memcpy`s, Telekine inserts a `copy_out` into the ExecStream after the application kernel, which generates the data. The `copy_out` writes the data to the assigned tagged buffer, followed by the tag to signal to Telekine that the data is ready. Since libTelekine runs on the client, it has no way of knowing when the copy out has completed until the tagged buffer has been copied back, so it will retry the same GPU-to-CPU copy until the tag is correct corresponding to a complete copy. This check is represented by the PEEK operation on line 27 of Algorithm 2; libTelekine only dequeues the operation after verifying that the `copy_out` kernel did its work on line 32.

GPU-to-GPU data copies. Emerging hardware supports dedicated, high-bandwidth, cross-GPU communication links such as NVLink [Fol16]. NVLink improves cross-GPU data copy efficiency but does not change the fundamental communication mechanisms used in a GPU stack. Telekine currently implements GPU-to-GPU copies as two copies: one from the first GPU back to the

client and the second from the client to the second GPU. Direct GPU-to-GPU copies using NVLink would be far more efficient, but to be data oblivious, they would have to occur at a fixed rate. We leave this task for future work.

Discussion. The XferStream is carefully constructed so that it never synchronizes with the ExecStream. The XferStream contains DMA operations, which the OS can detect; if application kernels on the ExecStream occupy the GPU causing the encryption kernels on the XferStream—and transitively the DMAs—to wait, then the platform can learn some information about kernel execution times. There may still be leakage between the XferStream and the ExecStream because we cannot guarantee that kernels of the former will not interfere with the latter. However, we believe this leakage to be hard to exploit in practice, we have not seen it in any of our benchmarks, and we expect that future GPU features like strict priority [NVI18] or preemption [TGC+14] will allow Telekine to seal the leak.

5.4.3 Data movement example.

Figure 5.4 how Telekine transforms application commands into equivalent, data-oblivious commands on the ExecStream and XferStream. The application issues three commands: ❶ copy data to the GPU, ❷ launch a kernel to process that data, and ❸ copy the results of the computation out of the GPU back to the CPU.

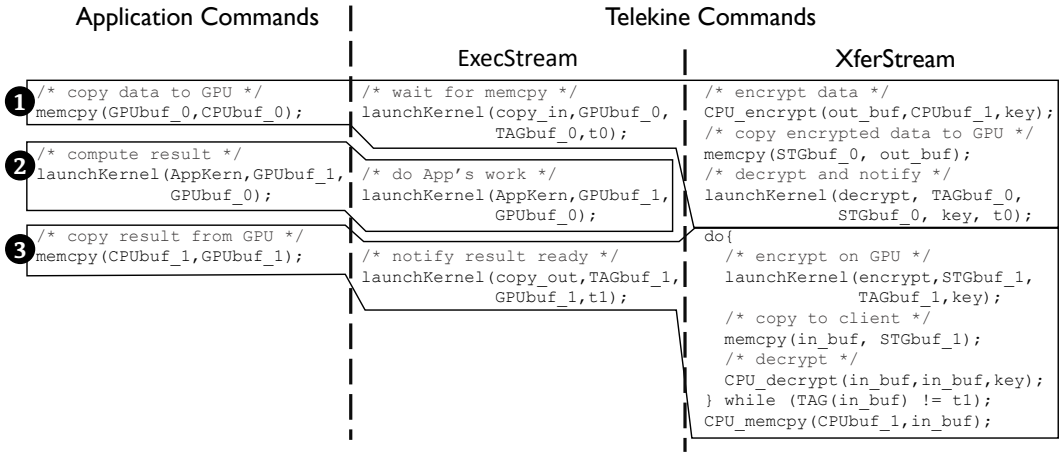

Figure 5.4: API calls made by the application and their mapping to underlying commands performed by Telekine.

❶ : The application requests a `memcpy` from `CPUbuf_0` to `GPUbuf_0`. In response, Telekine chooses a tag, `t0`, and tagged buffer, `TAGbuf_0`, for this operation. Then, it enqueues a kernel, `copy_in`, on the ExecStream. The `copy_in` kernel will spin on the GPU, using atomic operations to check the end of `TAGbuf_0` until it sees `t0`. Then it copies the contents of `TAGbuf_0` into `GPUbuf_0`. On the XferStream, Telekine encrypts the data, then copies the encrypted data to a staging buffer in untrusted GPU memory `STGbuf_0`. Finally, Telekine launches a kernel, `decrypt`, on the XferStream, which reads the encrypted data out of untrusted memory and decrypts it into `TAGbuf_0`. After the data is written, the tag `t0` is appended after a memory barrier, signaling to `copy_in` that the data is ready.

❷ : The application launches its kernel, `AppKern`, which processes the data in `GPUbuf_0` and writes its result into `GPUbuf_1`. Since `AppKern`

is launched on the ExecStream after `copy_in`, it will wait for `copy_in` to complete, ensuring that the data will be in `GPUbuf_0` before `AppKern` starts. The platform cannot detect that `AppKern` has started.

❸ : The application issues a request to copy the results of `AppKern` from `GPUbuf_1` to `CPU_buf1`. In response, Telekine again chooses a tag and tagged buffer, `t1`, and `TAGbuf_1`, respectively, and immediately enqueues a `copy_out` kernel on the ExecStream. After the application's kernel, `AppKern`, has completed, `copy_out` moves the result of its computation in `GPUbuf_1` into `TAGbuf_1` then atomically appends `t1`. While waiting for `copy_out` to finish, Telekine periodically encrypts `TAGbuf_1` into a staging buffer in untrusted memory, `STGbuf_1` then issues a `memcpy` operation to copy the contents of `STGbuf_1` to a client-side buffer, `in_buf`. Telekine decrypts `in_buf` and checks the tag. If the tag matches `t1`, `copy_out` and `AppKern` must have completed, and the data can be copied into `CPUbuf_1`. If not, this process will be repeated during the next scheduled GPU to client transfer.

5.4.4 Synchronizing data-oblivious streams

Applications sometimes wish to synchronize with their GPU streams (i.e., wait for all outstanding commands to complete), or synchronize one GPU stream with another (i.e., ensure another stream has completed some operation, n, before this stream starts operation, m). Telekine handles both of these cases by injecting kernels that increment a counter in GPU memory between kernels in the ExecStream.

The increment kernel only runs after all previous kernels in the stream, providing an accurate count of how many application kernels have executed because of stream semantics. Telekine copies that counter back to the client periodically and can block the application thread until all submitted work has completed.

5.5 Implementation

The Telekine prototype is based on AMD's ROCm 1.8 [AMD], an open-source software stack for AMD GPUs. Telekine requires an open-source stack because we split its functionality between user and cloud machines. NVIDIA is generally thought to have higher hardware and software performance as well as better third-party software support. But NVIDIA only officially supports closed-source drivers and runtimes.

LibTelekine and the relay. All applications were ported to use HIP [HIP], the ROCm CUDA replacement. LibTelekine marshals the arguments of HIP API calls before sending them over a TLS protected TCP connection to the relay to support initialization. The libTelekine and relay prototype are use code generated by AvA [YPAR20]; they total 8,843 and 5,650 lines of C/C++/HIP code, respectively (measured by cloc [clo]).

GPU TEE. GPU TEE requirements are made explicit in Section 3.3.2, and most of those requirements are safety properties that do not impact perfor-

	ExecStream		XferStream		
Benchmark	Quantum	Size	Quantum	Size	Bandwidth
Microbench	15ms	32kerns	30ms	1MB	533 Mb/s
MXNet	15ms	512kerns	30ms	1MB	533 Mb/s
Galois1	15ms	32kerns	30ms	1MB	533 Mb/s
Galois2	15ms	32kerns	30ms	1MB	533 Mb/s

Table 5.2: Data-oblivious schedule parameters and the network bandwidth required. MicroBench from §5.6.1; MXNet from §5.6.2; Galois1 executes on one GPU, Galois2 on two from §5.6.3. ExecStream sizes are the number of kernel launches, each of which is 320 bytes. XferStream streams contribute twice their size to bandwidth consumption because Telekine copies data in both directions at every quantum.

mance. A notable exception is the cryptography required to secure the secrecy and integrity of kernel launch commands. We model the timing of these features by decrypting kernel launch commands in the relay.

5.6 Evaluation

We quantify the overheads of the security Telekine provides by comparing it to an insecure baseline: applications run on cloud provider machines that offload computation to GPUs directly through the GPU runtime.

We measure Telekine across two testbeds. The first is the *simulated testbed*, which simulates the wide-area network (WAN) latencies and bandwidth, providing a controlled environment for measurement. The second is the *geodist testbed* in which the server and client are geo-distributed and connected by the Internet. Both testbeds use the same "cloud machine" (the *server*), which has an Intel i9-9900K CPU with eight cores @3.60GHz, 32GB

of RAM, and two Radeon RX VEGA 64 GPUs each with 8GB of RAM. All machines are running Ubuntu 16.04.6 LTS with Linux kernel version 4.13.0, and AMD's ROCm-1.8 runtime and HIP-1.5 compiler.

In the simulated testbed, the client has an Intel Xeon E3-1270 v6 processor with four cores @3.8GHz and 32GB of RAM. Both this client and the server have a Gtek X540 10Gb NIC, which we connect directly. We simulate a client-to-cloud network connection in a controlled environment using `netem` [net19], which allows us to add network delays and limit bandwidth. We always limit the bandwidth of the connection to 1Gbps, and unless otherwise mentioned, we add delays in both directions so that the total round trip time (RTT) is 10ms. These parameters are conservative for a network connection to an edge cloud server [YHQL15, CP17].

In the geodist testbed, the client is a VM hosted by vultr [Vul] in their Dallas, TX datacenter (the server is in Austin, TX). The VM has eight vCPUs and 32GB of RAM. We measured the RTT between the server and this client at 12ms, and the average bandwidth at 877Mbps.

Different applications use different schedules to get good performance, though Table 5.2 shows strong similarity among the data-oblivious schedules we use for evaluation.

5.6.1 Telekine performance tradeoff

Figure 5.5 shows the performance tradeoff for a microbenchmark with 16MB of input and output and a GPU kernel with a configurable running time

	ResNet	InceptionV3	DenseNet
Model size	97.5 MB	90.9 MB	30.4 MB
Input size			
Input image	224x224x3	299x299x3	224x224x3
Batch size	64	64	48
Data size per batch	9.2 MB	16.4 MB	6.9 MB
Single-GPU training baseline			
T-put	20.27 MB/s	11.05 MB/s	13.57 MB/s
T-put (less sync)	22.69 MB/s	11.66 MB/s	17.46 MB/s

Table 5.3: Overview of machine learning training on MXNet. The input size is given in pixel dimensions, batch size in images per GPU. T-put is throughput.

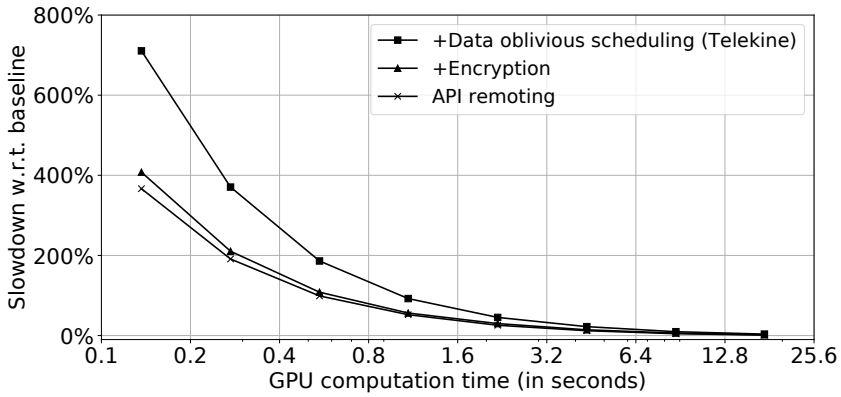

Figure 5.5: A microbenchmark that shows how Telekine overheads decrease as the running time of the GPU computation increases.

on the simulated testbed. The different lines show the costs of specific sources of overhead. The "API remoting" line uses the XferStream and the ExecStream over the network. The "+Encryption" line adds encryption to API remoting. Finally, the "Data-oblivious scheduling" line adds the data-oblivious schedule described in Table 5.2 to encryption. When the GPU kernel executes for only 0.14 seconds, the overhead of Telekine is nearly 8×. Once the computation takes 4.4s, the overhead is only 22%. Telekine is a remote execution system; it makes communication more expensive because of its oblivious scheduling as well as network delay and limited bandwidth. It is most efficient when computation dominates communication, which is the case for our benchmarks.

5.6.2 Machine learning algorithms

We port MXNet [CLL+15], a state-of-the-art machine learning library, to run on the HIP runtime. Our port is based on MXNet v1.1.0 (git commit `07a83a03`). We also use AMD's MIOpen library for efficient neural network operators. Some parts of MXNet adaptively choose from different GPU kernel implementations by measuring execution times on the available hardware and choosing the most performant option. To ensure the baseline and Telekine are running the same kernels for measurement purposes, we record the kernels chosen by the baseline and hard-code those kernel choices for all runs.

Optimizing MXNet. We applied several optimizations to MXNet, which help to mitigate the fact that Telekine is communicating with the GPU over

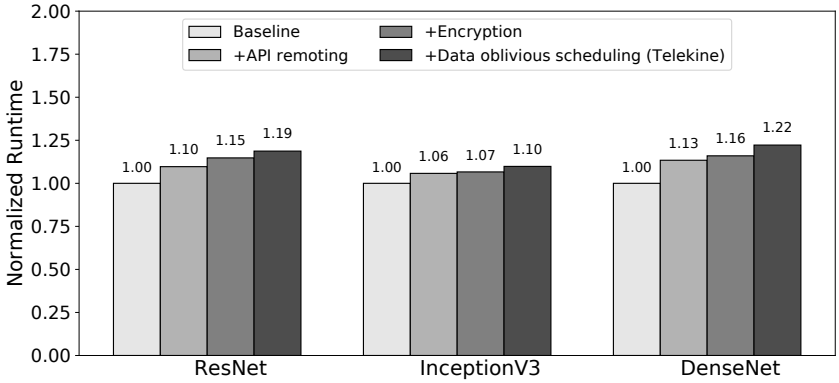

Figure 5.6: Performance of machine learning training algorithms using a single GPU with Telekine on the simulated testbed.

ResNet	InceptionV3	DenseNet
1.23×	1.08×	1.20×

Table 5.4: Performance of machine learning training algorithms on Telekine, measured on the geodist testbed.

a WAN:

- The models we evaluate represent the pixel channels of the input bitmaps using 4-byte floating point quantities, even though they range in integer values from 0 to 255. To save network bandwidth, we send bytes instead of floats, reducing bandwidth by 4×. Bytes are changed back floats on the GPU.

- We determined that MXNet was overly conservative in its GPU synchronization strategy and were able to reduce the number of synchronizations it performs by removing unnecessary calls to `hipStreamSyncronize` ("less sync" in Table 5.3). Telekine also optimizes synchronization calls by using tagged

buffers (§5.4.1) to coordinate data transfers.

Machine learning training. We evaluate the training performance of deep neural networks on Telekine using three state-of-the-art convolutional neural network architectures: ResNet [HZRS16], InceptionV3 [SVI+16], and DenseNet [HLWvdM17]. All models are trained using the ImageNet dataset (a substantial data set consisting of 1.4 million training images). For ResNet, we use the 50-layer variant. For DenseNet, we use the 121-layer variant. We evaluated all networks using batches size of 64. Table 5.3 summarizes the input sizes that were used to evaluate the three network architectures.

Figure 5.6 shows the performance of training three neural nets on Telekine using the simulated testbed, normalized to the insecure baseline. The bars break down Telekine's overheads and match the descriptions from Section 5.6.1. Both Telekine and the baseline use a single GPU. Table 5.4 shows the same experiment on the geodist testbed; the results are similar to the simulated testbed.

Machine learning inference. We evaluate neural network inference workloads for ResNet, InceptionV3, and DenseNet with Telekine. For inference, latency is the priority for users, but throughput is still a priority for providers. Batching inference can substantially improve throughput by fully utilizing hardware capabilities and amortizing the overheads from other system components [CWZ+17]. We evaluate the latency of inference with different batch

Batch size	ResNet Base	ResNet Telekine	InceptionV3 Base	InceptionV3 Telekine	DenseNet Base	DenseNet Telekine
Simulated testbed						
1	20	273 (13.7x)	29	259 (8.93x)	26	248 (9.54x)
8	42	270 (6.43x)	65	264 (4.06x)	47	241 (5.13x)
64	233	389 (1.67x)	368	559 (1.52x)	246	405 (1.65x)
256	988	1195 (1.21x)	1520	1806 (1.19x)	946	1163 (1.23x)
Geodist testbed						
1	20	200 (10.0x)	31	205 (6.61x)	26	201 (7.73x)
8	69	241 (3.49x)	111	247 (2.23x)	84	209 (2.49x)
64	462	481 (1.04x)	637	685 (1.08x)	484	483 (1.00x)

Table 5.5: Latencies (in ms) of machine learning inference workloads with the baseline system (Base in the Table) and Telekine.

sizes, ranging from 1 to 256. Our baseline is an insecure server with one local GPU, communicating with them over the network. Table 5.5 shows the inference latency of three neural networks with different batch sizes. The overheads with on the simulated testbed for batches of size 256 are 21%, 19%, and 23% for ResNet, InceptionV3, and DenseNet, respectively, which are slightly improved compared to the overheads we report for training (§5.6.2), although the training batch size was 64. With a batch size of 64, the overheads on the simulated testbed inflate to 67%, 52%, and 65%. When we move to the geodist testbed, the baseline's performance suffers more than Telekine; at batches of size 64, the standard deviation of our measurements exceeds the differences between the mean Telekine and baseline runs. Clipper [CWZ+17] uses an adaptive batch size to meet the application's latency requirement, which Telekine could adopt.

Application		Normalized runtime
BFS	(1 GPU)	1.18x
SSSP	(1 GPU)	1.21x
Pagerank	(1 GPU)	1.29x
BFS	(2 GPUs)	1.38x
SSSP	(2 GPUs)	1.41x

Table 5.6: Performance of Galois applications with Telekine.

5.6.3 Graph algorithms

Galois is a framework designed to accelerate parallel applications with irregular data access patterns, such as graph algorithms [PP16]. We port Galois's GPU computation to use the HIP runtime instead of CUDA and evaluate it on three graph algorithms: breadth-first search (BFS), PageRank, and single-source shortest paths (SSSP). All measurements use the USA roads graph dataset [DIM05]. Figure 5.6 shows the performance of these applications on Telekine with one and two GPUs. The baseline is an unmodified system with local GPU(s). Baseline performance for single GPU applications is BFS 54.1s, SSSP 74.6s, Pagerank 60.9s; for two GPUs: BFS 36.4s, SSSP 42.8s. For the input distributed with Galois, two GPU Pagerank slows down, so we do not evaluate it.

Telekine imposes moderate overheads on single-GPU Galois applications, adding latency to data transfer times. Galois implements each graph algorithm as a single GPU kernel that is iteratively called until the algorithm reaches termination. Multi-GPU applications exchange data between GPUs through the host after each iteration. Telekine imposes higher overheads for

RTT (ms)	ResNet	InceptionV3	DenseNet
10	1.19x	1.10x	1.22x
20	1.29x	1.13x	1.37x
30	1.44x	1.16x	1.49x
40	1.53x	1.18x	1.66x
50	1.62x	1.30x	2.09x

Table 5.7: Normalized runtime of machine learning workloads with respect to network round trip time (RTT).

multi-GPU workloads because of increased data movement over the network.

5.6.4 WAN latency sensitivity

Telekine assumes that the client communicates with the server over a WAN. The greater distances crossed by WANs result in longer round trip times (RTTs). The batching of commands that Telekine does for security also makes it resilient to these increased RTTs, especially when the ratio of GPU computation to communication is high. To demonstrate this, we increased the RTT between our machines using `netem` [net19] and ran the machine learning training benchmarks for different RTTs (Table 5.7). Overheads increase with RTT. At 30ms, which we measured to be the RTT between the client and an Amazon EC2 instance, the overhead for InceptionV3 is still only 16%.

Chapter 6

Related work

Using computational resources without trusting privileged software is an active area of research. Here we provide a survey of related work to provide context to work described in this dissertation. We start with a survey of shielding systems relevant to both Telekine and Ryoan, then cover work that is relevant to each system independently.

6.1 Shielding systems.

Shielding systems are designed to protect secret data while it is being processed in an untrusted environment. Unlike other shielding systems, Ryoan defends against the untrusted environment and also confines the application so that it need not be trusted to maintain data secrecy. No shielding system besides Telekine, to our knowledge, focusses on the communication issues that arise when shielding GPU communication from an untrusted platform.

6.1.1 Software shielding.

Software shielding uses a hypervisor or compiler to preserve the privacy and integrity of applications executing on an untrusted platform. Overshadow [CGL+08], InkTag [HKD+13], and Sego [KDL+16] use a trusted hy-

pervisor to protect trusted applications from an untrusted operating system. InkTag and Sego allow a trusted application to verify untrusted operating system services (e.g., a file system) with help from the hypervisor. Virtual Ghost [CDA14] uses a trusted compiler rather than a hypervisor for protection.

6.1.2 Hardware shielding.

Hardware shielding uses hardware primitives (such as SGX) to protect applications from platform software. Haven [BPH15], Scone [ATG+16], and Graphene-SGX [TPV17] allow a trusted program and its library operating system to execute in an SGX enclave that protects them from attack by host software. VC3 [SCF+15] secures trusted MapReduce using SGX.

Opaque [ZDB+17] uses carefully designed TEE code inside SGX enclaves to prevent leakage through known SGX side channels (e.g., memory access patterns). These techniques cannot be applied to code that is untrusted since the secret data owner would have to verify that they are in use. Opaque also deals with communication leakage but does not consider communication with GPUs.

ARM TrustZone [Lim] is another commercially available hardware primitive that protects computations from platform software. TrustZone provides a single "secure world," which allows code to execute in multiple privilege levels; in contrast, SGX provides an unlimited number of enclaves, all of which execute at user-level. TrustZone does not currently encrypt memory, so it is less

resistant to physical attacks, but TrustZone can deliver page faults to privileged code in the secure world, eliminating controlled channel attacks [XCP15]. Komodo [FBHP17] uses formally verified software to provide an enclave abstraction on top of TrustZone. In order to replace SGX with TrustZone, Ryoan would require a management layer like Komodo.

Trusted Platform Modules Attempts to use late launch and Trusted Platform Modules (TPMs) for user assurance (e.g., Flicker [MPP+08]) suffer from poor usability due to the restricted execution environment required by the TPM. Late-launched code has no access to the operating system and must manage the bare machine. Code executing in an enclave can be more complex than what is practical to execute in late launch.

Ironclad [HHL+14] addresses the limitations of the late launch environment with a (small) verified system stack that must be included with each trusted binary. Ironclad is not backward compatible and requires users to write verified code, placing a burden on the programmer.

MiniBox [LMN+14] uses a TPM and Native Client to protect an application and the OS from each other. Unlike Ryoan, MiniBox uses Native Client strictly to protect the OS and its secure hypervisor, not to prevent applications from leaking sensitive data.

For all TPM-based systems, a computation's data is visible on the memory bus, where an unscrupulous administrator of the host platform can

steal it. SGX enclave data is encrypted before it travels across the memory bus, preserving an enclave's secrecy.

6.1.3 Cryptographic shielding.

Homomorphic encryption [Gen09, BV11] allows untrusted code to compute directly on encrypted data with strong security guarantees. Unfortunately, practical implementations of general-purpose homomorphic encryption are not available, and current overheads are prohibitive.

Property-preserving encryption (for instance, order-preserving encryption [BCLO09]) can protect the secrecy of some computations [NKW15], and some systems use these primitives [PRZB11, BPTG15, SLPR15]. However, these systems have weaker security guarantees [GRS17], apply to limited scenarios, or have a significant performance overhead. In comparison, Ryoan's confinement does not require domain-specific knowledge about the applications. However, Ryoan does require stronger assumptions, i.e., that hardware and the Ryoan runtime are correct.

6.2 Timing and termination channels

Both Ryoan and Telekine are concerned with limiting information leakage through timing and termination channels. Timing and termination channels are studied in previous work [KWH11, For10] in the context of information flow control. In Ryoan, a module has to terminate for each unit of work, and the processing-time channel can only be used once per unit; different units will

not interfere due to module reset. In Telekine the end-to-end execution time of the application is leaked, but it is only leaked once and we found the end-to-end time to be a much weaker signal than the execution time of individual GPU kernels.

OS-level time protection. Recent extensions to seL4 [GYCH19] suggest general OS-level techniques that prevent timing-based covert channels by eliminating the sharing of hardware resources that can form the basis of covert channels. These techniques are not adequate to prevent malicious code from modulating its behavior time purposefully leak secrets, although they do lower their bandwidth. The techniques do not yet generalize to I/O-attached accelerators.

6.3 Work related to Ryoan: decentralized information flow control

Decentralized information flow control (DIFC) allows untrusted applications to access secret data but prevents them from leaking data to unauthorized parties. However, most DIFC systems require that all trusted code is deployed in a centralized platform or administrative domain under a trusted, privileged reference monitor [KYB+07,VEK+07,PBR+14,ZBwKM06,LGV+09, AGL+12]; similar enforcements have also been realized in a browser (COWL [SYM+14]) and a mobile device (Maxoid [XW15]). Two exceptions are DStar [ZBWM08] and Fabric [LGV+09], which do not have a centralized reference

monitor. However, although a DStar or Fabric user does not need to trust all machines involved in the system, they must trust the machine on which they process their data, which means a correct reference monitor (the OS or runtime that supports DIFC) must be properly installed on the machine, and that the machine's administrator does not use root privilege to steal secret data. Such trust is not required in Ryoan.

Systems that track information flow down to the hardware-gate level [TOL+11, TLW+09, LKO+14, ZWSM15] form a basis for strong information flow guarantees, and close timing and cache channels ignored by Ryoan. However, such hardware is not available and, as designed, does not include the privacy and integrity guarantees provided by SGX.

6.4 Work related to Telekine: secure computation on GPUs

Trusted Execution Environments on GPUs. HIX [JTK+19] extends an SGX-like design with duplicate versions of the enclave memory protection hardware to enable MMIO access from code running in an SGX enclave. This enables HIX to guarantee that a single enclave has exclusive access to the MMIO regions exported by a GPU, in principle, defeating a malicious OS that wants to interpose or create its own mappings to them. While this design provides stronger GPU isolation than current enclaves, it remains vulnerable to side-channel attacks because communication is not data oblivious.

Graviton [VVB18] supports GPU TEEs based on *secure contexts* that

use the GPU command processor to protect memory from other concurrently executing contexts. Similar to Telekine, Graviton secures communication using cryptographic techniques. Telekine can adopt many of Graviton's clever mechanisms for its TEE functionality (§3.3.2), such as restricting access to GPU page tables without trusting the kernel driver. But Graviton does not protect against side channels, which is Telekine's primary mission.

The opportunity to provide stronger security for GPU-accelerated applications using TEEs and oblivious communication has been observed by others [HJM+19].

Securing accelerators. SUD emulates a kernel environment in user space to isolate malicious device drivers [BWZ10]. Previous work has explored techniques to support trusted I/O paths, leveraging hypervisor support [WW17, ZGNM12] or system management mode [KKJ+16]. Our work focuses on the secure use of GPUs with untrusted system software and does not rely on support from the software at lower privilege layers. Border Control [OPHW15] addresses security challenges for accelerator-based systems but focuses on protecting the system from a malicious accelerator rather than Telekine, which protects CPU and GPU code from an untrusted platform.

GPU security and protection. Studies have analyzed GPU security properties and vulnerabilities [ZKR+17]. Frigo et al. [FGBR18] demonstrate techniques that leverage integrated GPUs to accelerate side-channel attacks from

browser codes using JavaScript and WebGL. PixelVault [VAPI14] exploits physical isolation between CPUs and GPUs to implement secure storage for keys, though it was shown to be insecure [ZKR+17]. CUDA Leaks [PLV16] shows techniques to exfiltrate data from the GPU to a malicious user. Attacks that take advantage of GPU memory reuse without re-initialization are a common theme [LKKK14,ZDL+17,HLH+17]. Several systems have proposed mechanisms that bring the GPU under tighter control of system software, exploring OS support [RCS+11,KLRI11,GST+11,MSS14], access to OS-managed resources [SFKW13,SFKW14,KHH+14], hypervisor support [TDC14,SKYK14, DS09, GGS+09, SCS09, GMAC10, VSB14] and GPU architectural support for cross-domain protection [ALM+17, CFHR17, PHW14, PHB14, VBO+16].

Secure machine learning. Ohrimenko et al. describe an SGX-based system for multi-party machine learning on an untrusted platform [OSF+16]. Their data-oblivious algorithm for convolutional neural networks explicitly does not support state-of-the-art operations that are data-dependent (e.g., max pooling). Telekine can support any data-dependent operations but requires a GPU TEE. Chiron [HSS+18] provides a framework for untrusted code to design and train machine learning models in SGX. Telekine does not support untrusted code but does allow the use of GPUs, which Chiron excludes. CQSTR [ZYC+16] lets a *trusted* platform operator confine untrusted machine learning code so that it can be securely applied to user data. By contrast, Telekine protects user data from an untrusted platform operator. MLcap-

sule [HZG+18] protects service provider secrets (machine learning model) and client data by running machine learning algorithms in an SGX enclave but does not suggest extensions to allow secure GPU acceleration.

Slalom [TB19] secures training of DNNs using a combination of TEEs and local GPUs. Slalom's guarantees are achieved by partitioning DNN training into linear layers using matrix multiplication, which is offloaded to a GPU, the remaining operators, which execute on the CPU in a TEE such as SGX. Matrix multiplication is verified and turned private using algorithmic techniques [Fre77], enabling secure GPU offload without requiring GPU TEE support.

Recent work [DGBL+16,LJLA17] demonstrates how to efficiently *apply* neural networks to encrypted data. As far as we know, today, there are no practical techniques for *training* deep neural networks on encrypted data.

API remoting. API remoting [DPS+11,RPS+12,LC11,KSL+12,BBNLS10, DIM+09, LNEAEG11, XBD+12] is an I/O virtualization technique that interposes a high-level user-mode API. API calls are forwarded to a user-level computing framework [SCS09] on a dedicated appliance VM [VSB14], or on a remote server [DPS+11, KSL+12]. To our knowledge, Telekine is the first system to use API remoting as a security technique.

Chapter 7

Conclusion

Hardware-protected TEEs, augmented with proper techniques from system software, are a promising step towards secure computation on untrusted public clouds. While the techniques described here and TEEs themselves certainly have their limitations, their combination represents a point in the space that achieves meaningful security at a reasonable cost. Both Ryoan and Telekine achieve their security goals with reasonable overheads: generally under 50% for the workloads that we measured, but of course, the actual overheads depend very much on the application and the data being processed.

Ryoan allows users to process data with software they do not trust, executing on a platform they do not control safely, thereby benefiting users, data processing services, and computational platforms. Ryoan does this by confining untrusted application code via a trusted sandbox (provided by Google's NaCl) that is itself made tamperproof via hardware enclave-protected execution (provided by Intel's SGX). Ryoan also defines and enforces an execution model that allows mutually distrustful software nodes to exchange data without disclosing secrets to each other or the platform provider. We implement and evaluate a Ryoan prototype over various case studies of real-world applications. Our evaluation, based on real SGX hardware and simulation, shows

that Ryoan overhead is workload-dependent, 27% in the best case, and up to 419% in the worst case.

Telekine enables secure GPU acceleration in the cloud. Telekine protects in-cloud computation with a GPU TEE and application/library computation by placing it on a client machine. It secures their communication with a novel GPU stream abstraction that ensures the execution is independent of input data. Telekine allows GPU-accelerated workloads such as training machine learning models to leverage cloud GPUs while providing strong secrecy and integrity guarantees that protect the user from the platform's privileged software and its administrators.

It is true that absolute performance is and will continue to be the most important factor for the majority of public cloud users. Viewed through that lens, any system with impacts performance in any amount is a non-starter. But the work presented here achieves security that was only possible previously with orders of magnitude of overhead (if you agree that our trust hardware is valid). We hope that this massive reduction in overhead will open up public clouds to more security-conscious users as they weigh public cloud deployments against investing in their own hardware.

Bibliography

[23aa] 23andMe Compares Family History and Genetic Tests for Predicting Complex Disease Risk. http://mediacenter.23andme.com/blog/23andme-compares-family-history-and-genetic-tests-for-predicting-complex-disease-risk/. (Accessed: September 2016).

[23ab] 23andMe. 23andMe. https://23andme.com/. Accessed: June 20, 2020.

[AGL+12] Owen Arden, Michael D George, Jed Liu, K Vikram, Aslan Askarov, and Andrew C Myers. Sharing mobile code securely with information flow control. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2012.

[ALM+17] Rachata Ausavarungnirun, Joshua Landgraf, Vance Miller, Saugata Ghose, Jayneel Gandhi, Christopher J. Rossbach, and Onur Mutlu. Mosaic: A GPU memory manager with application-transparent support for multiple page sizes. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, MICRO'17. IEEE, 2017.

[Amaa] Amazon. Amazon EC2 P3 Instances. https://aws.amazon.

com/ec2/instance-types/p3/. (Accessed: September 2018).

[Amab] Amazon Web Services. Amazon Web Services. https://aws.amazon.com/. Accessed: June 20, 2020.

[AMD] AMD. ROCm, a New Era in Open GPU Computing. https://rocm.github.io/index.html. (Accessed: February 12, 2020).

[aml] Amazon Machine Learning. https://aws.amazon.com/machine-learning/. (Accessed: September 2017).

[ANB+18] Marc Andrysco, Andres Nötzli, Fraser Brown, Ranjit Jhala, and Deian Stefan. Towards Verified, Constant-time Floating Point Operations. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 1369–1382, 2018.

[ATG+16] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, David Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure Linux Containers with Intel SGX. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16. USENIX Association, 2016.

[aws] User guide for Windows instances. https://docs.aws.amazon.

com/AWSEC2/latest/WindowsGuide/elastic-gpus.html. (Accessed: April 2019).

[AZM10] Aslan Askarov, Danfeng Zhang, and Andrew C. Myers. Predictive Black-box Mitigation of Timing Channels. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 297–307, New York, NY, USA, 2010. ACM.

[azu14] Azure: Microsoft's Cloud Platform, 2014. http://www.azure.microsoft.com.

[BBNLS10] A. Barak, T. Ben-Nun, E. Levy, and A. Shiloh. A package for OpenCL based heterogeneous computing on clusters with many GPU devices. In *2019 IEEE International Conference on Cluster Computing Workshops and Posters*, CLUSTER WORKSHOPS, September 2010.

[BCD+18] Andrea Biondo, Mauro Conti, Lucas Davi, Tommaso Frassetto, and Ahmad-Reza Sadeghi. The guard's dilemma: Efficient code-reuse attacks against intel SGX. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1213–1227, Baltimore, MD, August 2018. USENIX Association.

[BCLO09] Alexandra Boldyreva, Nathan Chenette, Younho Lee, and Adam O'neill. Order-preserving symmetric encryption. In *Annual In-*

ternational Conference on the Theory and Applications of Cryptographic Techniques (EuroCrypt), 2009.

[Bit] Bitfusion: The Elastic AI Infrastructure for Multi-Cloud. `https://bitfusion.io`. (Accessed: February 12, 2020).

[Bot] Lèon Bottou. Stochastic Gradient SVM. `http://leon.bottou.org/projects/sgd#stochastic_gradient_svm`. (Accessed: September, 2016).

[BPH15] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. *ACM Trans. Comput. Syst.*, 33(3):8:1–8:26, August 2015.

[BPTG15] Raphael Bost, Raluca Ada Popa, Stephen Tu, and Shafi Goldwasser. Machine Learning Classification over Encrypted Data. In *Network and Distributed System Security Symposium (NDSS)*, 2015.

[BV11] Zvika Brakerski and Vinod Vaikuntanathan. Fully homomorphic encryption from ring-LWE and security for key dependent messages. In *Advances in Cryptology (CRYPTO)*. 2011.

[BWZ10] Silas Boyd-Wickizer and Nickolai Zeldovich. Tolerating Malicious Device Drivers in Linux. In *Proceedings of the 2010 USENIX Annual Technical Conference*, USENIXATC'10. USENIX Association, 2010.

[CCX+18] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. SgxPectre Attacks: Leaking Enclave Secrets via Speculative Execution. *CoRR*, February 2018. `http://arxiv.org/abs/1802.09085`.

[CD16] Victor Costan and Srinivas Devadas. *Intel SGX Explained*. 2016.

[CDA14] John Criswell, Nathan Dautenhahn, and Vikram Adve. Virtual Ghost: Protecting Applications from Hostile Operating Systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[CFHR17] Jason Cong, Zhenman Fang, Yuchen Hao, and Glenn Reinman. Supporting Address Translation for Accelerator-Centric Architectures. In *IEEE International Symposium on High Performance Computer Architecture*, HPCA. IEEE, 2017.

[CGL+08] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffery Dwoskin,

and Dan R. K. Ports. Overshadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[cla] Clarifai. `https://www.clarifai.com`. (Accessed: September 2016).

[CLD16] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *Proceedings of the 25th USENIX Security Symposium (USENIX Security 16)*, pages 857–874, Berkeley, CA, USA, 2016. USENIX Association.

[CLL+15] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *CoRR*, abs/1512.01274, 2015. `http://arxiv.org/abs/1512.01274`.

[CLL+17] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Acoustic Cryptanalysis. *Journal of Cryptology*, 30, April 2017.

[clo] cloc: Count Lines of Code. `https://github.com/AlDanial/cloc`. (Accessed: February 12, 2020).

[CP17] Richard Cziva and Dimitrios P Pezaros. On the Latency Benefits of Edge NFV. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS. IEEE, 2017.

[CPG+04] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding data lifetime via whole system simulation. In *USENIX Security Symposium*, 2004.

[CS13] Stephen Checkoway and Hovav Shacham. Iago Attacks: Why The System Call API Is a Bad Untrusted RPC Interface. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, number CS2012-0984, July 2013.

[CVDBDS09] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2009.

[CWZ+17] Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. Clipper: A Low-latency Online Prediction Serving System. In *Proceedings of the 14th*

USENIX Conference on Networked Systems Design and Implementation, NSDI'17. USENIX Association, 2017.

[CZRZ17] Sanchuan Chen, Xiaokuan Zhang, Michael K Reiter, and Yinqian Zhang. Detecting privileged side-channel attacks in shielded execution with Déjá Vu. In *ACM Symposium on Information, Computer and Communications Security (AsiaCCS)*, 2017.

[DDS+09] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. In *The Conference on Computer Vision and Pattern Recognition*, CVPR. IEEE, 2009.

[DGBL+16] Nathan Dowlin, Ran Gilad-Bachrach, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy. In *International Conference on Machine Learning*, 2016.

[DIM05] DIMACS. 9th DIMACS Implementation Challenge - Shortest Paths. `http://users.diag.uniroma1.it/challenge9/download.shtml`, 2005. (Accessed: February 12, 2020).

[DIM+09] José Duato, Francisco D Igual, Rafael Mayo, Antonio J Peña, Enrique S Quintana-Ortí, and Federico Silla. An efficient implementation of GPU virtualization in high performance clusters.

In *European Conference on Parallel Processing*, Euro-Par'09, pages 385–394, Berlin, Heidelberg, 2009. Springer, Springer-Verlag.

[dis] The DisGeNET Database. `http://www.disgenet.org/ds/DisGeNET/files/current/DisGeNET_2016.db.gz`. (Accessed: February, 2016).

[DMS04] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th USENIX Security Symposium*, USENIX Security 04, pages 21–21, Berkeley, CA, USA, 2004. USENIX Association.

[DPS+10] J. Duato, AJ. Pena, F. Silla, R. Mayo, and E.S. Quintana-Orti. rCUDA: Reducing the Number of GPU-Based Accelerators in High Performance Clusters. In *2010 International Conference on High Performance Computing Systems*, HPCS, 2010.

[DPS+11] Jose Duato, Antonio J. Pena, Federico Silla, Juan C. Fernandez, Rafael Mayo, and Enrique S. Quintana-Orti. Enabling CUDA acceleration within virtual machines using rCUDA. In *Proceedings of the 2011 18th International Conference on High Performance Computing*, HIPC '11, pages 1–10, Washington, DC, USA, 2011. IEEE Computer Society.

[DS09] Micah Dowty and Jeremy Sugerman. GPU virtualization on

VMware's hosted I/O architecture. *ACM SIGOPS Operating Systems Review*, 43(3):73–82, 2009.

[Dwo07] Morris Dworkin. NIST Special Publication 800-38D: Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. `http://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf`, 2007. (Accessed: February 12, 2020).

[ema] The Radicati Group, Inc: Email Statistics Report 2009-20013 (summary). `http://www.radicati.com/wp/wp-content/uploads/2009/05/email-stats-report-exec-summary.pdf`. (Accessed: September 2016).

[FBHP17] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using Verification to Disentangle Secure-enclave Hardware from Software. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 287–305, New York, NY, USA, 2017. ACM.

[FGBR18] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU. In *IEEE Symposium on Security and Privacy*, May 2018.

[Fol16] Denis Foley. Ultra-Performance Pascal GPU and NVLink Interconnect. In *HotChips*, 2016.

[For10] Bryan Ford. Plugging Side-Channel Leaks with Timing Information Flow Control. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2010.

[Fre77] Rusins Freivalds. Probabilistic Machines Can Use Less Running Time. In *IFIP Congress*, pages 839–842, 1977.

[FRK02] Hubertus Franke, Rusty Russell, and Matthew Kirkwood. Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux. In *Ottawa Linux Symposium*, 2002.

[FWZ+16] Andrew Ferraiuolo, Yao Wang, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. Lattice Priority Scheduling: Low-Overhead Timing-Channel Protection for a Shared Memory Controller. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2016.

[Gen09] Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. `https://crypto.stanford.edu/craig`.

[GESM17] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache Attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security*, EuroSec'17, 2017.

[GGKSC13] Scott Grauer-Gray, William Killian, Robert Searles, and John Cavazos. Accelerating Financial Applications on the GPU. In

Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units, GPGPU-6, pages 127–136, New York, NY, USA, 2013. ACM.

[GGS+09] Vishakha Gupta, Ada Gavrilovska, Karsten Schwan, Harshvardhan Kharche, Niraj Tolia, Vanish Talwar, and Parthasarathy Ranganathan. GViM: GPU-accelerated virtual machines. In *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing*, pages 17–24. ACM, 2009.

[GMAC10] Giulio Giunta, Raffaele Montella, Giuseppe Agrillo, and Giuseppe Coviello. A GPGPU transparent virtualization component for high performance computing clouds. In *European Conference on Parallel Processing*, pages 379–391. Springer, Springer, 2010.

[Goo] Google. Google Cloud. `https://cloud.google.com/`. Accessed: June, 2020.

[GRS17] Paul Grubbs, Thomas Ristenpart, and Vitaly Shmatikov. Why your encrypted database is not secure. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, pages 162–168, New York, NY, USA, 2017. ACM.

[GST+11] Vishakha Gupta, Karsten Schwan, Niraj Tolia, Vanish Talwar, and Parthasarathy Ranganathan. Pegasus: Coordinated Scheduling for Virtualized Accelerator-based Systems. In *Proceedings*

of the 2011 USENIX Conference on USENIX Annual Technical Conference, USENIXATC'11, pages 3–3. USENIX Association, 2011.

[GYCH19] Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. Time Protection: The Missing OS Abstraction. In *European Conference in Computer Systems*, EuroSys, 2019.

[Hem17] Nicole Hemsoth. Medical Imaging Drives GPU Accelerated Deep Learning Developments. `https://www.nextplatform.com/2017/11/27/medical-imaging-drives-gpu-accelerated-deep-learning-developments/`, November 2017. (Accessed: February 12, 2020).

[HHL+14] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad Apps: End-to-End Security via Automated Full-System Verification. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[HIP] HIP: Convert CUDA to Portable C++ Code. `https://github.com/ROCm-Developer-Tools/HIP`. (Accessed: February 12, 2020).

[HJM+19] Tyler Hunt, Zhipeng Jia, Vance Miller, Christopher J. Rossbach, and Emmett Witchel. Isolation and Beyond: Challenges

for System Security. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, pages 96–104, New York, NY, USA, 2019. ACM.

[HJM+20] Tyler Hunt, Zhipeng Jia, Vance Miller, Ariel Szekely, Yige Hu, Christopher J. Rossbach, and Emmett Witchel. Telekine: Secure Computing with Cloud GPUs. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 817–833, Santa Clara, CA, February 2020. USENIX Association.

[HKD+13] Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. InkTag: Secure Applications on an Untrusted Operating System. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[HLH+17] Ari B. Hayes, Lingda Li, Mohammad Hedayati, Jiahuan He, Eddy Z. Zhang, and Kai Shen. GPU Taint Tracking. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, pages 209–220. USENIX Association, 2017.

[HLWvdM17] Gao Huang, Zhuang Liu, Kilian Q Weinberger, and Laurens van der Maaten. Densely connected convolutional networks.

In *Proceedings of the IEEE conference on computer vision and pattern recognition*, volume 1, page 3, 2017.

[HSS+18] Tyler Hunt, Congzheng Song, Reza Shokri, Vitaly Shmatikov, and Emmett Witchel. Chiron: Privacy-preserving Machine Learning as a Service. *CoRR*, abs/1803.05961, 2018. `http://arxiv.org/abs/1803.05961`.

[HZG+18] Lucjan Hanzlik, Yang Zhang, Kathrin Grosse, Ahmed Salem, Max Augustin, Michael Backes, and Mario Fritz. MLCapsule: Guarded Offline Deployment of Machine Learning as a Service. *CoRR*, abs/1808.00590, 2018. `http://arxiv.org/abs/1808.00590`.

[HZRS16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[HZX+16] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 533–549. USENIX Association, 2016.

[HZX+18] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. *ACM Trans. Comput. Syst.*, 35(4):13:1–13:32, December 2018.

[ibm] IBM Visual Recognition service. `http://www.ibm.com/smarterplanet/us/en/ibmwatson/developercloud/visual-recognition.html`. (Accessed: September 2016).

[Int] Intuit. TurboTax. `https://turbotax.com/`. Accessed: June 20, 2020.

[Int14] Intel(R) Software Guard Extensions Programming Reference. `https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf`, 2014. (Accessed: February 12, 2020).

[JDK+16] Prerit Jain, Soham Desai, Seongmin Kim, Ming-Wei Shih, Jae-Hyuk Lee, Changho Choi, Youjung Shin, Taesoo Kim, Brent Byunghoon Kang, and Dongsu Han. OpenSGX: An Open Platform for SGX Research. In *Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2016.

[JFK16] Zhen Hang Jiang, Yunsi Fei, and David Kaeli. A complete key recovery timing attack on a GPU. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2016.

[JFK17] Zhen Hang Jiang, Yunsi Fei, and David Kaeli. A Novel Side-Channel Timing Attack on GPUs. In *Proceedings of the on Great Lakes Symposium on VLSI 2017*, GLSVLSI '17, pages 167–172, New York, NY, USA, 2017. ACM.

[JLLK17] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. Sgx-bomb: Locking down the processor via rowhammer attack. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, SysTEX'17, New York, NY, USA, 2017. Association for Computing Machinery.

[Joh17] Simon Johnson. Intel SGX and Side-Channels. `https://software.intel.com/en-us/articles/intel-sgx-and-side-channels`, March 2017. (Accessed: February 12, 2020).

[JTK+19] Insu Jang, Adrian Tang, Taehoo Kim, Simha Sethumadhavan, and Jaehyuk Huh. Heterogeneous Isolated Execution for Commodity GPUs. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'19, 2019.

[KDK+14] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. *SIGARCH Comput. Archit. News*, 42(3):361–372, June 2014.

[KDL+16] Youngjin Kwon, Alan Dunn, Michael Lee, Owen Hofmann, Yuanzhong Xu, and Emmett Witchel. Sego: Pervasive Trusted Metadata for Efficiently Verified Untrusted System Services. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[KGG+18] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. *CoRR*, abs/1801.01203, January 2018. `http://arxiv.org/abs/1801.01203`.

[KHH+14] Sangman Kim, Seonggu Huh, Yige Hu, Xinya Zhang, Emmett Witchel, Amir Wated, and Mark Silberstein. GPUnet: Networking Abstractions for GPU Programs. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 201–216. USENIX Association, 2014.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '99, pages 388–397, Berlin, Heidelberg, 1999. Springer-Verlag.

[KKJ+16] Yonggon Kim, Ohmin Kwon, Jinsoo Jang, Seongwook Jin, Hyeongboo Baek, Brent Byunghoon Kang, and Hyunsoo Yoon. On-demand bootstrapping mechanism for isolated cryptographic operations on commodity accelerators. 62, 7 2016.

[KLRI11] Shinpei Kato, Karthik Lakshmanan, Raj Rajkumar, and Yutaka Ishikawa. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *Proc. USENIX ATC*, USENIX-ATC'11, pages 17–30. USENIX Association, 2011.

[KMPS11] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation. In *NDSS*, 2011.

[KPMR12] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. STEALTHMEM: system-level protection against cache-based side channel attacks in the cloud. In *USENIX Security Symposium*, 2012.

[KSL+12] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee. SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters. In *Proceedings of the 26th ACM international conference on Supercomputing*, pages 341–352. ACM, 2012.

[KWH11] Vineeth Kashyap, Ben Wiedermann, and Ben Hardekopf. Timing- and termination-sensitive secure information flow: Exploring a new approach. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2011.

[KYB+07] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans, Kaashoek Eddie, and Kohler Robert Morris. Information flow control for standard OS abstractions. In *ACM Symposium on Operating System Principles (SOSP)*, 2007.

[Lam73] Butler W. Lampson. A Note on the Confinement Problem. *Communications of the ACM(CACM)*, 16(10), October 1973.

[LC11] Tyng-Yeu Liang and Yu-Wei Chang. GridCuda: A Grid-Enabled CUDA Programming Toolkit. In *Advanced Information Networking and Applications (WAINA), 2011 IEEE Workshops of International Conference on*, pages 141–146, March 2011.

[LCW13] Anyi Liu, Jim Chen, and Harry Wechsler. Real-time covert timing channel detection in networked virtual environments. In *International Conference on Digital Forensics*, 2013.

[LGV+09] Jed Liu, Michael D George, Krishnaprasad Vikram, Xin Qi, Lucas Waye, and Andrew C Myers. Fabric: A platform for secure distributed computation and storage. In *ACM Symposium on Operating System Principles (SOSP)*, 2009.

[LHH+15] Chang Liu, Michael Hicks, Austin Harris, Mohit Tiwari, Martin Maas, and Elaine Shi. GhostRider: A Hardware-Software System for Memory Timerace Oblivious Computation. In *Inter-*

national Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2015.

[lib] libsodium: A modern and easy-to-use crypto library. `https://github.com/jedisct1/libsodium`. (Accessed: September 2016).

[Lim] Arm Limited. Introducing Arm TrustZone. `https://developer.arm.com/technologies/trustzone`. (Accessed: February 12, 2020).

[LJLA17] Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. Oblivious Neural Network Predictions via MiniONN transformations. Cryptology ePrint Archive, Report 2017/452, 2017. `http://eprint.iacr.org/2017/452`.

[LKC+18] Dayeol Lee, David Kohlbrenner, Kevin Cheang, Cameron Rasmussen, Kevin Laeufer, Ian Fang, Akash Khosla an Chia-Che Tsai, Sanjit Seshia, Dawn Song, and Krste Asanovic. Keystone Enclave: An Open-Source Secure Enclave for RISC-V. `https://keystone-enclave.org/files/keystone-risc-v-summit.pdf`, 2018. (Accessed: February 12, 2020).

[LKKK14] Sangho Lee, Youngsok Kim, Jangwoo Kim, and Jong Kim. Stealing Webpages Rendered on Your Browser by Exploiting

GPU Vulnerabilities. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 19–33, Washington, DC, USA, 2014. IEEE Computer Society.

[LKO+14] Xun Li, Vineeth Kashyap, Jason K. Oberg, Mohit Tiwari, Vasanth Ram Rajarathinam, Ryan Kastner, Timothy Sherwood, Ben Hardekopf, and Frederic T. Chong. Sapper: A language for hardware-level security policy enforcement. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 97–112, New York, NY, USA, 2014. ACM.

[LKS+20] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An Open Framework for Architecting Trusted Execution Environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys'20, New York, NY, USA, 2020. Association for Computing Machinery.

[LMN+14] Yanlin Li, Jonathan McCune, James Newsome, Adrian Perrig, Brandon Baker, and Will Drewry. MiniBox: A Two-Way Sandbox for x86 Native Code. In *USENIX Anual Technical Conference*, number CMU-CyLab-14-001, 2014.

[LNEAEG11] Teng Li, Vikram K Narayana, Esam El-Araby, and Tarek El-Ghazawi. GPU resource sharing and virtualization on high per-

formance computing systems. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 733–742. IEEE, 2011.

[LSG+17] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *USENIX Security Symposium*, 2017.

[LSG+18] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Dkaniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *CoRR*, abs/1801.01207, January 2018. `http://arxiv.org/abs/1801.01207`.

[LWL15] Fangfei Liu, Hao Wu, and Ruby B. Lee. Can randomized mapping secure instruction caches from side-channel attacks? In *Hardware and Architectural Support for Security and Privacy*, 2015.

[MAA+16] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos Rozas. Intel Software Guard Extensions (Intel SGX) Support for Dynamic Memory Management Inside an Enclave. In *Hardware and Architectural Support for Security and Privacy*, 2016.

[ML97] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *ACM Symposium on Operating System Principles (SOSP)*, 1997.

[mos] Moses. http://www.statmt.org/moses/. (Accessed: September 2016).

[MP] Jay Mahadeokar and Gerry Pesavento. Open Sourcing a Deep Learning Solution for Detecting NSFW Images. https://yahooeng.tumblr.com/post/151148689421/open-sourcing-a-deep-learning-solution-for. (Accessed: September 2016).

[MPP+08] Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. In *ACM European Conference in Computer Systems (EuroSys)*, April 2008.

[MRR+15] Ramya Jayaram Masti, Devendra Rai, Aanjhan Ranganathan, Christian Müller, Lothar Thiele, and Srdjan Capkun. Thermal Covert Channels on Multi-core Platforms. In *USENIX Security Symposium*, 2015.

[MSS14] Konstantinos Menychtas, Kai Shen, and Michael L. Scott. Disengaged Scheduling for Fair, Protected Access to Fast Computational Accelerators. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 301–316, New York, NY, USA, 2014. ACM.

[MV05] David A. McGrew and John Viega. The Galois/Counter mode of operation (GCM), 2005.

[nac] Implementation and safety of NaCl SFI for x86-64. https://groups.google.com/forum/#!topic/native-client-discuss/C-wXFdR2lf8. (Accessed: September 2016).

[net19] netem. https://wiki.linuxfoundation.org/networking/netem, 2019. (Accessed: February 12, 2020).

[NKW15] Muhammad Naveed, Seny Kamara, and Charles V. Wright. Inference attacks on property-preserving encrypted databases. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 644–655, New York, NY, USA, 2015. ACM.

[NNQAG18] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. Rendered Insecure: GPU Side Channel Attacks Are Practical. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[NVIa] NVIDIA. CUDA Zone. https://developer.nvidia.com/cuda-zone. (Accessed: February 12, 2020).

[NVIb] NVIDIA. Driving Innovation: Building AI-Powered Self-Driving Cars. https://www.nvidia.com/en-us/self-driving-cars/. (Accessed: February 12, 2020).

[NVIc] NVIDIA. RISC-V Story. https://riscv.org/wp-content/

uploads/2016/07/Tue1100_Nvidia_RISCV_Story_V2.pdf. (Accessed: February 12, 2020).

[NVI16] NVIDIA. GPUs and DSLs for Life Insurance Modeling. https://devblogs.nvidia.com/gpus-dsls-life-insurance-modeling/, March 2016. (Accessed: February 12, 2020).

[NVI17a] NVIDIA. Microsoft Sets New Speech Recognition Record. https://news.developer.nvidia.com/microsoft-sets-new-speech-recognition-record/, August 2017. (Accessed: February 12, 2020).

[NVI17b] NVIDIA. NVIDIA CUDA Toolkit Documentation. http://docs.nvidia.com/cuda/cuda-runtime-api/stream-sync-behavior.html, 2017. (Accessed: February 12, 2020).

[NVI18] NVIDIA. CUDA Toolkit Documentation (Streams). https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#streams, 2018. (Accessed: February 12, 2020).

[OMA+18] Dan O'Keeffe, Divya Muthukumaran, Pierre-Louis Aublin, Florian Kelbert, Christian Priebe, Josh Lind, Huanzhou Zhu, and Peter Pietzuch. spectre-attack-sgx, 2018.

[OPHW15] Lena E. Olson, Jason Power, Mark D. Hill, and David A. Wood. Border Control: Sandboxing Accelerators. In *Proceedings of the*

48th International Symposium on Microarchitecture, MICRO-48, pages 470–481, New York, NY, USA, 2015. ACM.

[OSF+16] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Sebastian Nowozin Aastha Mehta, Kapil Vaswani, and Manuel Costa. Oblivious Multi-Party Machine Learning on Trusted Processors. In *USENIX Security Symposium*, 2016.

[OTK+18] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. Varys: Protecting SGX enclaves from practical side-channel attacks. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, pages 227–240. USENIX Association, 2018.

[PBR+14] Donald E. Porter, Michael D. Bond, Indrajit Roy, Kathryn S. McKinley, and Emmett Witchel. Practical Fine-Grained Information Flow Control Using Laminar. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 37(1), 2014.

[PHB14] Bharath Pichai, Lisa Hsu, and Abhishek Bhattacharjee. Architectural Support for Address Translation on GPUs: Designing Memory Management Units for CPU/GPUs with Unified Address Spaces. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'14, 2014.

[PHW14] Jonathan Power, Mark D Hill, and David A Wood. Supporting x86-64 Address Translation for 100s of GPU Lanes. In *HPCA*, 2014.

[Pix] Pixlr. Pixlr - Photo editor online. `https://pixlr.com/`. Accessed: June 20, 2020.

[PLV16] Roberto Di Pietro, Flavio Lombardi, and Antonio Villani. CUDA Leaks: A Detailed Hack for CUDA and a (Partial) Fix. *ACM Trans. Embed. Comput. Syst.*, 15(1):15:1–15:25, January 2016.

[PP16] Sreepathi Pai and Keshav Pingali. A Compiler for Throughput Optimization of Graph Algorithms on GPUs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 1–19, New York, NY, USA, 2016. ACM.

[PRZB11] Raluca Ada Popa, Catherine Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: protecting confidentiality with encrypted query processing. In *ACM Symposium on Operating System Principles (SOSP)*, 2011.

[PS19] Sandro Pinto and Nuno Santos. Demystifying Arm TrustZone: A Comprehensive Survey. *ACM Computing Surveys*, 51(6), 2019.

[qem] QEMU: open source processor emulator. `http://wiki.qemu.org/Main_Page`. (Accessed: September 2016).

[QS16] R. Qiao and M. Seaborn. A new approach for rowhammer attacks. In *2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 161–166, 2016.

[RCS+11] Christopher J. Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. PTask: Operating System Abstractions to Manage GPUs as Compute Devices. In *Symposium on Operating Systems Principles*, SOSP'11, pages 233–248. ACM, 2011.

[RFK+15] Ling Ren, Christopher W. Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. Constants Count: Practical Improvements to Oblivious RAM. In *USENIX Security Symposium*, 2015.

[RLT15] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *USENIX Security Symposium*, 2015.

[RPS+12] C. Reano, A. J. Pena, F. Silla, J. Duato, R. Mayo, and E. S. Quintana-Orti. CU2rCU: Towards the complete rCUDA remote GPU virtualization and sharing solution. *20th Annual International Conference on High Performance Computing*, 0:1–10, 2012.

[rtx18] GeForce RTX 2080 Ti, 2018. https://www.nvidia.com/en-us/geforce/graphics-cards/rtx-2080-ti/.

[SCF+15] Felix Schuster, Manuel Costa, Cedric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: Trustworthy Data Analytics in the Cloud using SGX. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2015.

[SCS09] Lin Shi, Hao Chen, and Jianhua Sun. vCUDA: GPU accelerated high performance computing in virtual machines. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, May 2009.

[SFB+15] Erik Smistad, Thomas L. Falch, Mohammadmehdi Bozorgi, Anne C. Elster, and Frank Lindseth. Medical image segmentation on GPUs - A comprehensive review. *Medical Image Analysis*, 20(1):1–18, 2015.

[SFKW13] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. GPUfs: Integrating a File System with GPUs. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, volume 32, March 2013.

[SFKW14] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel.

GPUfs: Integrating a File System with GPUs. *ACM Transactions on Computer Systems*, 32(1), 2014.

[sgxa] Intel(R) Software Guard Extensions for linux* OS, linux-sgx. `https://github.com/01org/linux-sgx`. (commit:d686fb0).

[sgxb] Intel(R) Software Guard Extensions for Linux∗OS, linux-sgx-driver. `https://github.com/01org/linux-sgx-driver`. (commit:0fb8995).

[sgx15] Intel Software Guard Extensions SDK for Linux OS. `https://download.01.org/intel-sgx/linux-1.9/docs/Intel_SGX_SDK_Developer_Reference_Linux_1.9_Open_Source.pdf`, 2015. (Accessed: September 2017).

[SKYK14] Yusuke Suzuki, Shinpei Kato, Hiroshi Yamada, and Kenji Kono. GPUvm: Why Not Virtualizing GPUs at the Hypervisor? In *USENIX ATC*, USENIX ATC'14, pages 109–120. USENIX Association, 2014.

[SLKP17] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *Network and Distributed System Security Symposium (NDSS)*, 2017.

[SLM+19] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. Zom-

bieLoad: Cross-Privilege-Boundary Data Sampling. *CoRR*, abs/1905.05726, 2019. http://arxiv.org/abs/1905.05726.

[SLPR15] Justine Sherry, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. Blindbox: Deep packet inspection over encrypted traffic. In *ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*, 2015.

[SMB+10] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. Adapting Software Fault Isolation to Contemporary CPU Architectures. In *USENIX Security Symposium*, 2010.

[spa] CSMINING Group: Spam Email Datasets. https://csmining.org/index.php/spam-email-datasets-.html. (Accessed: April 2016).

[SSR] Matthew J.A. Smith, Mikayel Samvelyan, and Tabish Rashid. Using AI to Solve Collaborative Challenges by Playing StarCraft. https://news.developer.nvidia.com/using-ai-to-solve-collaborative-challenges-by-playing-starcraft/. (Accessed: February 12, 2020).

[Sta] Statista. Amazon Web Services - Statistics & Facts. https://www.statista.com/topics/4418/amazon-web-services/. Accessed: June 20, 2020.

[SVI+16] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2818–2826, 2016.

[SYM+14] Deian Stefan, Edward Z Yang, Petr Marchenko, Alejandro Russo, Dave Herman, Brad Karp, and David Mazieres. Protecting users by confining JavaScript with COWL. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[TB19] Florian Tramè and Dan Boneh. Slalom: Fast, Verifiable and Private Execution of Neural Networks in Trusted Hardware. In *International Conference on Learning Representations*, ICLR '19, 2019.

[TDC14] Kun Tian, Yaozu Dong, and David Cowperthwaite. A Full GPU Virtualization Solution with Mediated Pass-through. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 121–132. USENIX Association, 2014.

[TGC+14] Ivan Tanasic, Isaac Gelado, Javier Cabezas, Alex Ramirez, Nacho Navarro, and Mateo Valero. Enabling Preemptive Multiprogramming on GPUs. In *ISCA*, 2014.

[TLW+09] Mohit Tiwari, Xun Li, Hassan M. G. Wassel, Frederic T. Chong, and Timothy Sherwood. Execution Leases: A Hardware-supported Mechanism for Enforcing Strong Non-interference. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009.

[TOL+11] Mohit Tiwari, Jason K. Oberg, Xun Li, Jonathan Valamehr, Timothy Levin, Ben Hardekopf, Ryan Kastner, Frederic T. Chong, and Timothy Sherwood. Crafting a Usable Microkernel, Processor, and I/O System with Strict and Provable Information Flow Security. In *International Symposium on Computer Architecture (ISCA)*, 2011.

[TPV17] Chia-Che Tsai, Donald E. Porter, and Mona Vij. Graphene-sgx: A practical library os for unmodified applications on sgx. In *Proceedings of the 2017 USENIX Annual Technical Conference*, USENIX ATC '17, pages 645–658, Berkeley, CA, USA, 2017. USENIX Association.

[VAPI14] Giorgos Vasiliadis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. PixelVault: Using GPUs for Securing Cryptographic Operations. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1131–1142, New York, NY, USA, 2014. ACM.

[VBMW+18] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security Symposium*, 2018.

[VBO+16] Jan Vesely, Arkaprava Basu, Mark Oskin, Gabriel H. Loh, and Abhishek Bhattacharjee. Observations and Opportunities in Architecting Shared Virtual Memory for Heterogeneous Systems. In *ISPASS*, 2016.

[VEK+07] Steve Vandebogart, Petros Efstathopoulos, Eddie Kohler, Maxwell Krohn, Cliff Frey, David Ziegler, Frans Kaashoek, Robert Morris, and David Mazières. Labels and Event Processes in the Asbestos Operating System. *ACM Transactions on Computer Systems (TOCS)*, 25(4), December 2007.

[VSB14] Lan Vu, Hari Sivaraman, and Rishi Bidarkar. GPU virtualization for high performance general purpose computing on the ESX hypervisor. In *Proceedings of the High Performance Computing Symposium*, page 2. Society for Computer Simulation International, 2014.

[vSGBR18] Stephan van Schaik, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Malicious Management Unit: Why Stopping Cache

Attacks in Software is Harder Than You Think. In *USENIX Security*, August 2018.

[Vul] Vultr.com. `https://www.vultr.com/products/cloud-compute/`. (Accessed: November 2019).

[VVB18] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. Graviton: Trusted Execution Environments on GPUs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[wmt] Shared Task: Machine Translation. `http://www.statmt.org/wmt13/translation-task.html`. (Accessed: September 2016).

[WW17] Samuel Weiser and Mario Werner. SGXIO: Generic Trusted I/O Path for Intel SGX. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, CODASPY '17, pages 261–268, New York, NY, USA, 2017. ACM.

[WX15] Zhenyu Wu and Zhang Xu. Whispers in the Hyper-Space: High-Bandwidth and Reliable Covert Channel Attacks Inside the Cloud. *IEEE/ACM Transactions on Networking*, 23(2), April 2015.

[XBD+12] Shucai Xiao, Pavan Balaji, James Dinan, Qian Zhu, Rajeev Thakur, Susan Coghlan, Heshan Lin, Gaojin Wen, Jue Hong,

and Wu-chun Feng. Transparent accelerator migration in a virtualized GPU environment. In *Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGrid, pages 124–131, 2012.

[XBJ+11] Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh Joshi, Matti Hiltunen, and Richard Schlilchting. An Exploration of L2 Cache Covert Channels in Virtualized Environments. In *ACM Workshop on Cloud computing security*, 2011.

[XCP15] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2015.

[XW15] Yuanzhong Xu and Emmett Witchel. Maxoid: Transparently Confining Mobile Applications with Custom Views of State. In *ACM European Conference in Computer Systems (EuroSys)*, 2015.

[YHQL15] Shanhe Yi, Zijiang Hao, Zhengrui Qin, and Qun Li. Fog Computing: Platform and Applications. In *ACM/IEEE Workshop on Hot Topics in Web Systems and Technologies*, HotWeb, 2015.

[YL08] Tatu Ylonen and Chris Lonvick. RFC 5246: The Transport Layer Security (TLS) Protocol: Version 1.2. `https://tools.`

ietf.org/html/rfc5246, August 2008. (Accessed: September 2016).

[YPAR20] Hangchen Yu, Arthur M. Peters, Amogh Akshintala, and Christopher J. Rossbach. AvA: Accelerated Virtualization of Accelerators. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[YSD+09] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2009.

[ZAM11] Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Predictive mitigation of timing channels in interactive systems. In *ACM Conference on Computer and Communications Security (CCS)*, 2011.

[ZAM12] Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Language-based Control and Mitigation of Timing Channels. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2012.

[ZBwKM06] Nickolai Zeldovich, Silas Boyd-wickizer, Eddie Kohler, and David Mazières. Making Information Flow Explicit in HiStar. In

USENIX Symposium on Operating Systems Design and Implementation (OSDI), pages 263–278. USENIX Association, 2006.

[ZBWM08] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazieres. Securing Distributed Systems with Information Flow Control. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2008.

[ZDB+17] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *USENIX Symposium on Networked Systems Design and Implementation*, NSDI, 2017.

[ZDL+17] Zhe Zhou, Wenrui Diao, Xiangyu Liu, Zhou Li, Kehuan Zhang, and Rui Liu. Vulnerable GPU Memory Management: Towards Recovering Raw Data from GPU. *PoPETs*, 2017(2):57–73, 2017.

[ZGNM12] Z. Zhou, V. D. Gligor, J. Newsome, and J. M. McCune. Building Verifiable Trusted Path on Commodity x86 Computers. In *2012 IEEE Symposium on Security and Privacy*, May 2012.

[ZJRR12] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-VM side channels and their use to extract private keys. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[ZKR+17] Zhiting Zhu, Sangman Kim, Yuri Rozhanski, Yige Hu, Emmett Witchel, and Mark Silberstein. Understanding The Security of Discrete GPUs. In *Proceedings of the General Purpose GPUs*, GPGPU-10, pages 1–11, New York, NY, USA, 2017. ACM.

[ZWC+13] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2013.

[ZWSM15] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. A hardware design language for timing-sensitive information-flow security. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 503–516, New York, NY, USA, 2015. ACM.

[ZYC+16] Yan Zhai, Lichao Yin, Jeffrey S Chase, Thomas Ristenpart, and Michael M Swift. CQSTR: Securing Cross-Tenant Applications with Cloud Containers. In *ACM Symposium on Cloud Computing*, 2016.

Vita

Tyler Scott Hunt was born in Durango, Colorado. After completing his work at Piedra Vista High School, Farmington, New Mexico, in 2010, he entered New Mexico State University in Las Cruces, New Mexico. He received the degree of Bachelor of Science from New Mexico State University in May 2013. In August 2013, he entered the doctoral program in the Department of Computer Science at the University of Texas at Austin.

Permanent address: tylerscotthunt@gmail.com

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.